UNIVERSITY OF WEST ATTICA
ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ

Department of Informatics and Computer Engineering

Master of Science
Cybersecurity

Malware Analysis and Reverse Engineering

Author: Chronis Anastasios - cscyb21033
Supervisor: Dr. Panayiotis Yannakopoulos

Athens, 2023

School of Engineering
Department of Informatics and Computer Engineering
Post-Graduate Studies Programme: CYBERSECURITY

M.Sc. Thesis
Malware Analysis and Reverse Engineering
.
Chronis Anastasios - cscyb21033

Supervisor: Panayiotis Yannakopoulos

Examination Committee:

Name	Signature
Panayiotis Yannakopoulos	Panagiotis Giannakopoulos Digitally signed by Panagiotis Giannakopoulos Date: 2023.08.24 08:09:10 +03'00'
Konstantinos Mavrommatis	Konstantinos Mavrommatis Digitally signed by Konstantinos Mavrommatis Date: 2023.08.25 14:53:04 +03'00'
Dimitrios Kogias	[illegible]

Abstract

Malware is a constantly growing threat to both individuals and organizations as it can be used to steal sensitive data, disrupt both minor and vital operations, and in some cases cause physical damage to hardware and even humans. Reverse engineering is a powerful tool for analyzing and understanding software, hardware and, in our case, malware. It allows analysts to disassemble and decompile the code to determine its functionality and identify vulnerabilities.

In this thesis, we present a comprehensive study of malware analysis and reverse engineering techniques. We will begin by setting up a safe Lab environment, capable of protecting the analyst while also providing them the tools needed to do their job.

The second part will be dedicated to Malware Analysis, starting with reviewing the history and types of malware, before delving deeper, with tools and techniques used in Static and Dynamic Analysis, code deobfuscation and closing with a concise workflow.

The third part will be reviewing Reverse Engineering tools and techniques as well as its importance in a malware analyst's repertoire.

We also discuss the ethical concerns of malware analysis and reverse engineering, like the legal issues surrounding the possession and distribution of malware, as well as the importance of a professional approach to the matter.

Finally, we will make a small statement about the future of Malware Analysis and Reverse Engineering and provide use cases that Machine Learning can be used to help the analysts secure an overall safer technological infrastructure.

Our work serves as an introduction point for cybersecurity analysts and computer or software engineers that want to dig deeper into malware analysis.

Acknowledgements

I would like to express my gratitude towards Dr. Panayiotis Yannakopoulos and the rest of the professors of the University of West Attica’s Cybersecurity Masters Program as well as my colleagues of the 2021 class. I would like to extend that gratitude to the entirety of the Cybersecurity community that keeps on providing free reading and practice material as well resources for the ones that want to join their ranks.
On a more personal note, I want to thank my mother, my sister and my friends for being at my side, my colleagues in the National Bank of Greece and especially my manager Nikolaos Kipos and my supervisor Dr. Emmanouil Ladakis, the local gym that is proven time and again to be a place of calm and solace, and finally, most of all, my father, who remains an inspiration almost 20 years after his passing.

Table of Contents

Lab Setup

Overview

In order to perform malware analysis, we first need to understand the dangers it inherently exposes us and our system to. We will be purposefully inflicting our system with malware. And on our system, we have stored our personal data and information, including passwords, credit cards, photographs, medical records and more. The last thing we want is for them to get somehow leaked to the general public, or to become a node of a widely infected network used for possibly illegal activities.
To circumvent these issues, we will use two virtual machines. There are two main benefits of virtual machines. First and foremost, we create a safe environment, that is separated from our host operating system that can be infected without us worrying about our main system. Second, and equally important, we can take snapshots of these virtual machines that will allow us to revert any changes and resume our work from a previous state that has either not been infected or has the malware at the state that we desire.
Two virtual machines will be created. The first one will be a Windows machine and it will have all the tools we will use to perform malware analysis and reverse engineering. The second one will be a Kali Linux machine that will be used to monitor the first machine's network traffic during the malware analysis and for reverse engineering.

Definitions and Specifications

The main system, from now on referred to as host, is the main computer that I use daily. It uses an Arch (BTW) Linux operating system and has 16GB of RAM and a 2013 4-core CPU (total 8 threads).
The first virtual machine, from now on referred to as Flare VM, uses Windows 10 as its base operating system. It will be given 2 cores (4 threads) and 8GB of RAM. More importantly, a lot of tools will be added that will be of use for the entire assignment. These tools are included in the FlareVM. FlareVM is a Windows distribution that is installed on top of the base operating system. It removes some key Windows features, like the Windows Update and the Microsoft Defender and that is on purpose since we will be infecting the machine with malware. FlareVM is developed by Mandiant (mandiant.com - last day visited: 14/2/2023). Flare will be infected with various malware throughout the analyst's career.
The second virtual machine, from now on referred to as Kali, uses Kali Linux, a GNU/Linux operating system based on Debian. It will be given 1 core (2 threads) and 4 GB of RAM. Kali Linux is probably the most well known operating system for security research, penetration testing, and digital forensics. It is developed by Offensive Security

(https://www.offensive-security.com/ - last day visited: 14/2/2023). Kali will be used as a gateway for Flare's internet traffic.

Notes: These are the specifications I will be using but are not mandatory or even important for that matter. Flare and Kali offer nothing more than a compilation of tools bundled in a convenient and easily installed package.
Additionally, the wide variety of hypervisors (QEMU in my case, but there are other cross-platform notable ones like VMWare and VirtualBox) can make them work in virtually every host operating system. And for advanced users, a type 1 hypervisor like Proxmox can make everything run with minimum overhead on a dedicated machine used as a Lab.
Analysts that focus more on static analysis or reverse engineering, can omit the Kali virtual machine. Doing so is obviously not as effective as the industry standard two machine setup. While monitoring the network can provide valuable insight on dynamic analysis, it does require more resources and that is why it can be ignored in some cases.
Finally, a case has been made for the Remnux Linux distribution (based on Ubuntu) for the gateway, with the argument that it is more Malware Analysis oriented than Kali. Remnux however is dorman with its last update more than two years ago, so it will not be preferred.

Lab Network

Network Topology

In order to monitor Flare's traffic effectively, we will need to change a few settings and route it through Kali first.

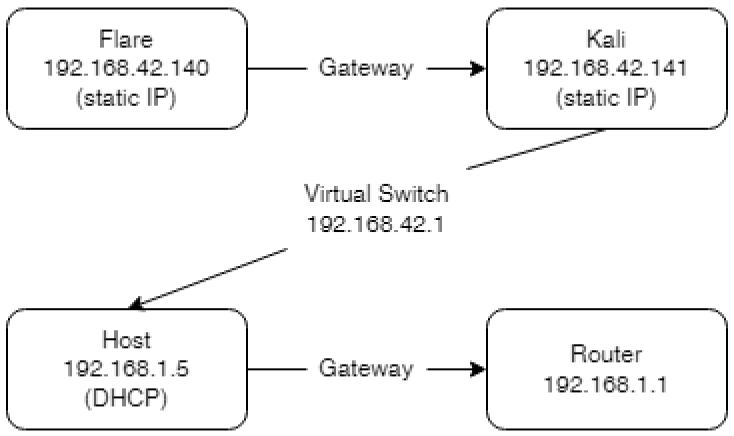
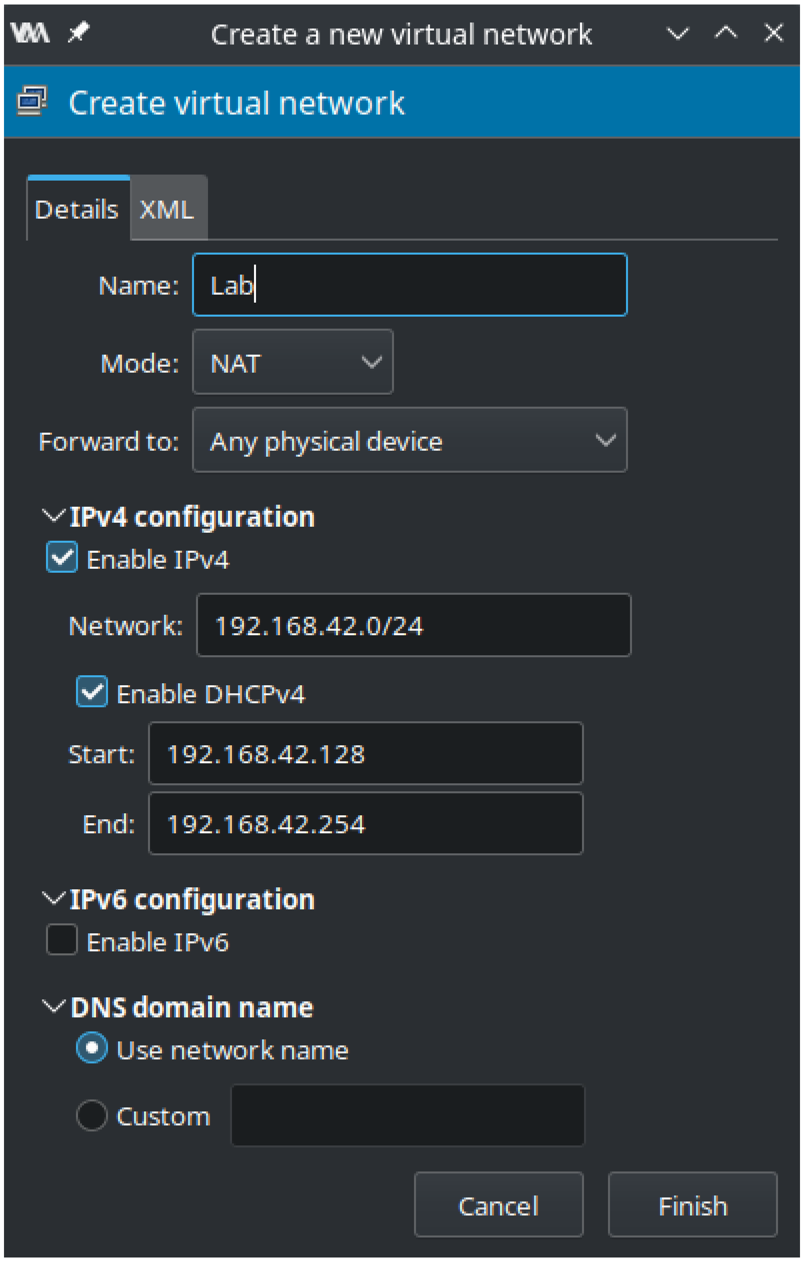


Figure 1.3.1 - Network Topology

Virtual Network Setup

First, we need to create the Virtual Network in the KVM manager and give it its separate IP range. To do this, we need to go to "Connection Details" under the Edit tab of the Virtual Machine Manager and add a new one using the plus sign. In this case, we will use NAT mode with the Network's address at 192.168.42.0 and available IPs from 192.168.42.128 up to 192.168.42.254.


Create a new virtual network

Create virtual network

Details XML

Name: Lab

Mode: NAT

Forward to: Any physical device

IPv4 configuration

[✓] Enable IPv4

Network: 192.168.42.0/24

[✓] Enable DHCPv4

Start: 192.168.42.128

End: 192.168.42.254

IPv6 configuration

[ ] Enable IPv6

DNS domain name

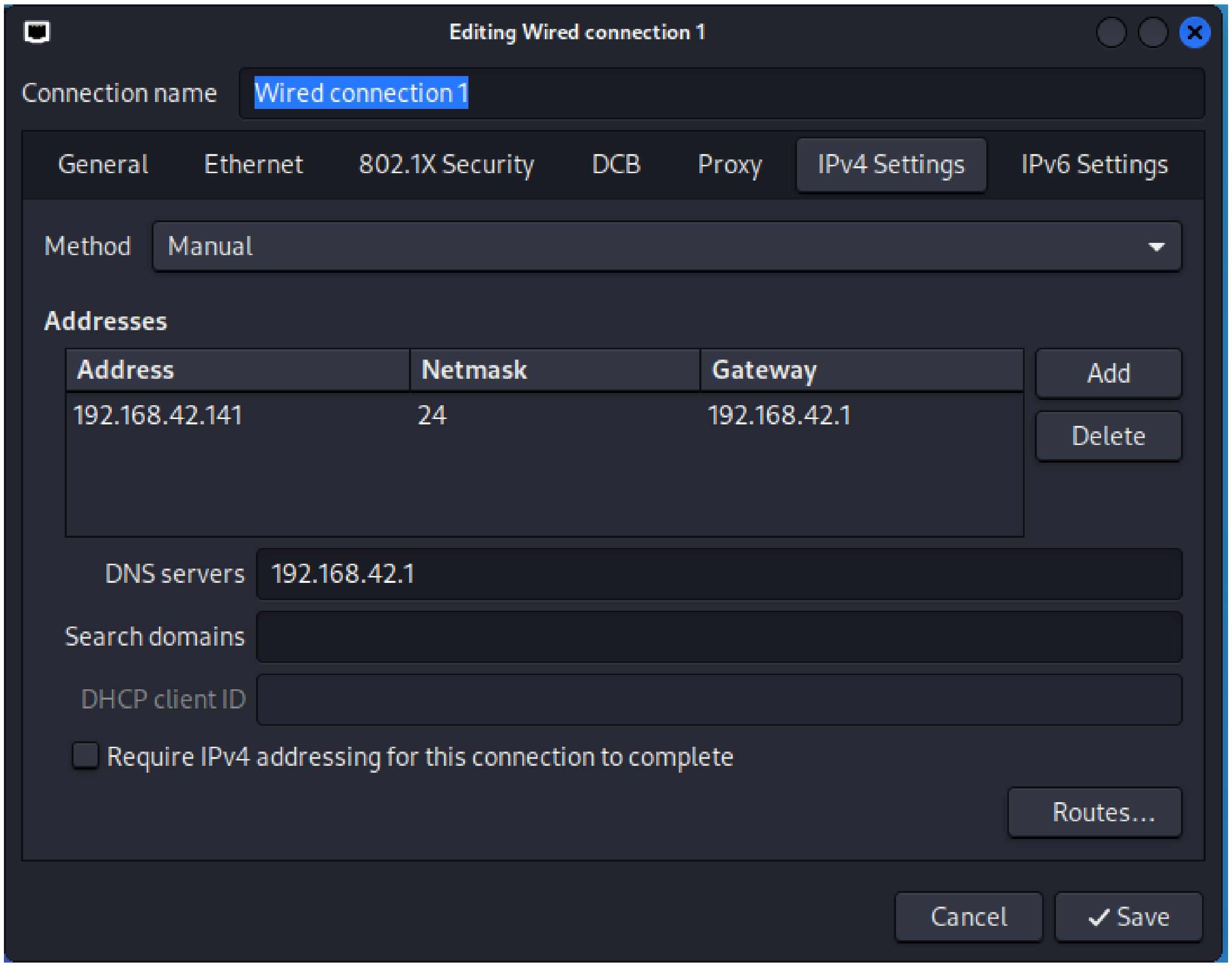
● Use network name

○ Custom

Figure 1.3.2 - Creating Virtual Network on Virtual Machine Manager

Gateway Setup

Now, we need to set Kali as the gateway. This procedure does require a few steps. First of all, we need to set a static IP. We can do that by opening the Advanced Network Configuration, selecting the “Wired connection 1” option and pressing at the cog to get into the configuration. We need to set the Method to Manual (from DHCP) and add the address. In this case, it will be the following.


Editing Wired connection 1

Connection name: Wired connection 1

General | Ethernet | 802.1X Security | DCB | Proxy | IPv4 Settings | IPv6 Settings

Method: Manual

Addresses

Address	Netmask	Gateway
192.168.42.141	24	192.168.42.1

DNS servers: 192.168.42.1

Search domains:

DHCP client ID:

Require IPv4 addressing for this connection to complete

Figure 1.3.3 - Setting up a static IP on Kali Linux

Then, we need to add a route for Flare on the Kali. To do so we need to open a terminal and type the following command:

sudo route add -net <Windows VM subnet> netmask <Windows VM subnet mask> gw <Kali Linux VM IP address>

In our case, this translates to:

```
route add -net 192.168.42.0 netmask 255.255.255.0 gw 192.168.42.141
```

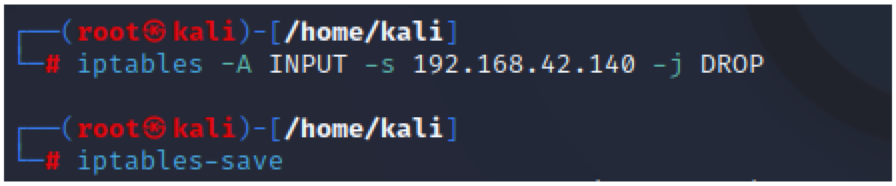
Finally, we need to set up the firewall for the Kali VM. To do so, we once again need to open the terminal. We will first need to install ufw. This is a user-friendly frontend for iptables. To install it, we type the command:

```
sudo apt update && sudo apt install ufw
```

Then we need to allow traffic. We can do that with the commands:

```
ufw enable
ufw default allow incoming
```

This does give access to the internet to Flare VM. However, we are studying malware here and sometimes, we only want to see the IP addresses, not actually communicate with them. In these cases, we can drop traffic incoming from Flare with this command:

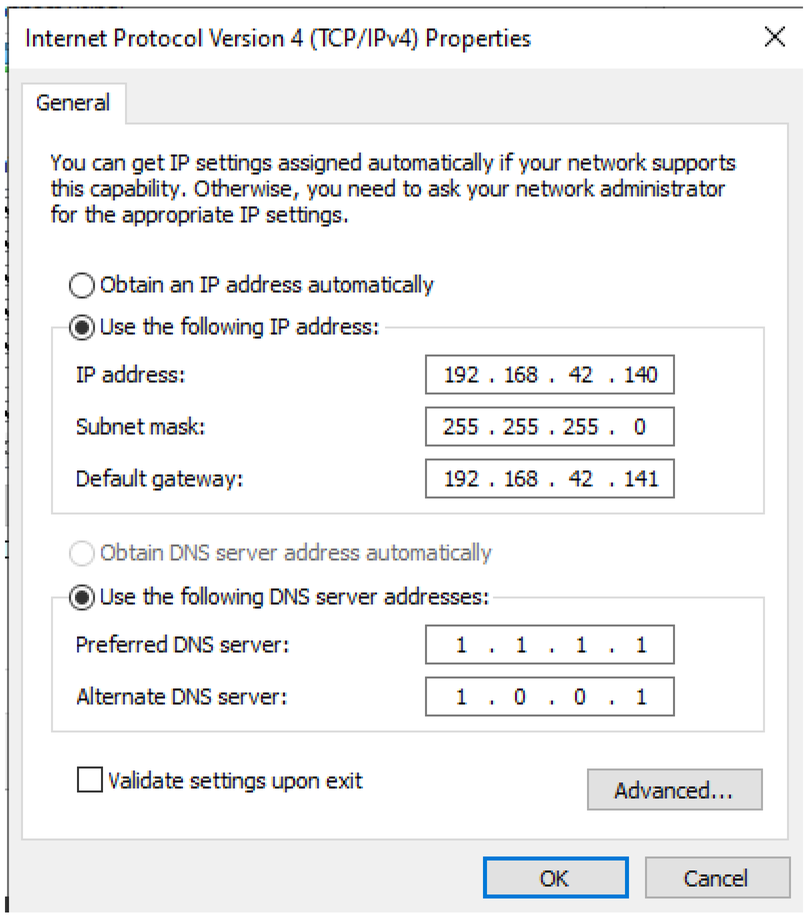


```
┌──(root㉿kali)-[/home/kali]
└─# iptables -A INPUT -s 192.168.42.140 -j DROP

┌──(root㉿kali)-[/home/kali]
└─# iptables-save
```

Figure 1.3.4 - Blocking outwards traffic from Flare VM

Flare VM Network Setup

In order to set the static IP at the Flare VM, we need to go to the Network Adapters. To do so, at the start menu, we search for “Network Status”, go to the “Ethernet” tab and then “Change adapter Options”. By right-clicking the adapter and going to the properties, we can edit the IPv4 settings to the following:



Internet Protocol Version 4 (TCP/IPv4) Properties

General

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

( ) Obtain an IP address automatically
[✓ Use the following IP address:]

IP address:	192 . 168 . 42 . 140
Subnet mask:	255 . 255 . 255 . 0
Default gateway:	192 . 168 . 42 . 141

( ) Obtain DNS server address automatically
[✓ Use the following DNS server addresses:]

Preferred DNS server:	1 . 1 . 1 . 1
Alternate DNS server:	1 . 0 . 0 . 1

[ ] Validate settings upon exit

Figure 1.3.4 - Setting up a static IP on Windows

Both VMs should have an internet connection now and all of Flare’s traffic is routed through Kali.

Malware Analysis

Malware: Definition, Types and a short History

Definition

Malware, a word deriving from Malicious Software, is an application designed to extract information, utilize resources, get access to a computer, remotely execute code or do whatever else a malicious actor wants to do with a computer that is not his own but has control over.

Types of Malware

There are several types of malware, distinguished by the way they operate and the way they affect various systems.

Viruses: A virus is a rather small program that often infects the target as part of legitimate software or file and then, once activated, starts making copies of itself. When the virus is run, it will probably try to infect the rest of the network either by automatically send data to other connected computers or by utilizing a more subtle approach and wait for the users to share them using social engineering techniques. Viruses is by far the most common type of malware and can cause a wide range of damage. It can delete files, steal personal information like passwords or bank details, encrypt or corrupt data, and even make the infected computer part of a botnet used to launch Distributed Denial of Service (DDoS) attacks.

Worms: Worms, just like viruses, will start replicating themselves, however it is software of its own and is not getting attached to other software or files. Worms usually exploit vulnerabilities in operating systems or other legitimate software like internet browsers. Once they infect a computer they will try to spread through the rest of the network. Worms are capable of spreading via email, instant messaging and social networks, but due to their nature of not being attached to a legitimate piece of software or file, they usually require a zero day vulnerability or a serious user error. Still, since they can give the hacker the option of executing code remotely, worms can be a devastating infection.

Trojan Horses: Trojan horses are types of malware that disguise themselves as legitimate software in order to trick the user into installing it. Most trojan horses are delivered via browser with fake websites imitating the official page of a free tool like 7zip, obs or audacity. Unlike worms that utilize vulnerabilities, trojan horses need the user to misjudge a situation, visit the fake webpage, download the malicious executable and actually run it on their computer. Then, the hacker can get a reverse shell, giving them full access to the user's computer.

Rootkits: Rootkits is a weird and divisive type of malware. Rootkits can be used to hide the presence of other malware on a system. However, there are companies like video games publishers and companies performing exams and issuing certificates that openly install rootkits on the user's computer as a cheating prevention procedure. And most users are fine with it, not knowing the implications. What's worse is that these rootkits are usually not removed when the user uninstalls the piece of software they came with. Rootkits can be very difficult to detect and remove. They often use stealth techniques to evade detection by antivirus software and can be used to give the malicious actors complete and unauthorized access to a computer. Then, the hacker can perform a wide variety of attacks to either the user, the entire network or even use the computer as a proxy for other malicious activities.

Spyware: Spyware is another divisive type of malware. It collects information of the user's activities and can steal personal information such as login credentials, credit card numbers, medical records and other sensitive or personal information. What makes spyware divisive among users is the fact that governments and security agencies around the world often use these types of malware in order to better monitor the population for illegal activities. And while cybersecurity professionals understand the dangers and take precautionary measures against such malware, some end users are in favor of them in the name of safety and security. They are wrong, but this will not stop some three letter agencies from invading citizen privacy.

Adware: Adware oftenly ends up in a computer without the user's knowledge because they did not disable a pre-checked agreement box in the installation process of some other software. It usually comes in the form of a browser extension or a toolbar or even some otherwise legitimate software like antivirus programs. Adware is no longer that common but was widely used in the past, even by very reputable companies, to force the user into a specific browser or search engine or to install the trial version of some software against the user's will. What's more, adware would often display pop-up windows of advertisements that were notoriously hard to close and often came with video and sound, prompting the user to subscribe to any type of overpriced services.

Cryptojacking: Cryptojacking is a rather newer type of malware. Instead of stealing information, passwords and credit cards, it installs a crypto miner at the victim’s computer and utilizes its resources to mine various types of cryptocurrency for the hacker. This is a serious offense for three major reasons. First of them, the victim gets a slower computer since the mining software drains most of the computer’s resources.

Second, the victim will have to pay a massive electricity bill at the end of the month and finally, the hardware gets degraded as constantly working at 100% can make it more susceptible to damage.

Ransomware: Ransomware is the final type of malware we will discuss and one of the most serious ones. Ransomware can affect everyone, from individual users to public services like hospitals and even to billion dollar corporations. Once infected, the ransomware will encrypt the victim's files, making them inaccessible. The hacker will then ask for payment, usually in the form of cryptocurrencies, in order to provide the decryption key. Most corporations have a bitcoin wallet and the main reason is to preemptively own one in case of a catastrophic ransomware attack. In the best case scenario, the victim will have a safe backup and get out of the situation virtually unscathed, but in the worst case scenario involves the victim paying the hackers and still not getting the decryption key.

History of Malware

The idea of software capable of replicating itself, was first conceptualized by John von Neumann in as early as 1949. Though just an idea at the time, it provided the foundation for the first malware created 22 years later, a virus called Creeper, originally written by Bob Thomas and later redesigned by Ray Tomlinson. Creeper could copy itself on a computer and used ARPANET (the predecessor of Internet) to spread throughout the entire network.
Ray Tomlinson later created the Reaper with the sole purpose of removing Creeper and its copies. Reaper is considered to be the first anti-virus. Creeper however was not really malicious, especially considering that the operators of the infected computers knew about it and how it was going to actually infect them. Much later, Tomlinson gave an interview[1](last visited - 26/01/2023) going in depth on the way Creeper and Reaper worked.

The first real malware was Wabbit, a virus created in 1974, capable of rapidly depleting a system's resources by constantly creating new processes and copying the original file. This procedure, known as a “fork bomb” due to the Operating System's fork mechanism, made the computer practically unusable and is the first case of a Denial of Service attack. The only weakness of Wabbit was that it could not spread via the network.

The first trojan horse was created in 1975 and was called PREVADE[2] (Internet source - last visited 26/01/2023). It would be called as a subroutine of the game named ANIMAL and would copy itself in every directory the user had access to. It would eventually spread by users trading the game on tapes.

In 1988 the Morris Internet Worm was created by Robert Tappan Morris and resulted with him being the first arrest since the release of malware was considered to be a felony in the United States under the 1986 Computer Fraud and Abuse act. The Morris Internet Worm was the first time that a malware far exceeded its original (non malicious) purpose and resulted in infecting around 10% of the internet at the time.

While in the subject of malware that run amok, in 2005 Samy Kamkar released Samy (also known as JS.Spacehero), a worm designed specifically for the MySpace social network that infected over 1 million users in less than 24 hours. While the intent was also not malicious, the worm brought down MySpace for a few days, just months after its acquisition by a major media outlet for hundreds of millions of dollars. Samy Kamkar pleaded guilty and, after completing his sentence, is now a well regarded security researcher. The Samy Worm was the first time the media and the general public understood the dangers and the complexity of cybersecurity and resulted in a major boost in research and resources allocated to the sector.

Finally, a more recent attack was the WannaCry ransomware that affected Windows systems that were not updated or past their official end-of-life. WannaCry infected more than 300.000 computers, including vital infrastructure like hospitals and power and water companies in more than 150 countries in just a few hours, before Marcus Hutchins found a kill switch. Digital forensics investigation indicated that the WannaCry originated from North Korea, despite the Eternal Blue exploit being designed by the NSA.

Malware Analysis: Definition and Types

Malware Analysis is the study, dissection and risk analysis of such programs. Its purpose is to allow researchers better understand the vulnerabilities that allow malicious actors to infect and take advantage of systems and facilitate the secure disclosure and eventually the update of these vulnerabilities.

There are three types of Malware Analysis

Static Analysis

Static Analysis is like trying to judge a christmas present from looking at the package. A bicycle wrapped in simple christmas paper is obviously a bicycle. Static Analysis is doing exactly that. Looking at the outside layers of the executable files without actually detonating the malware. In the following chapter we are going to see what information we can extract and how to actually extract it.

Dynamic Analysis

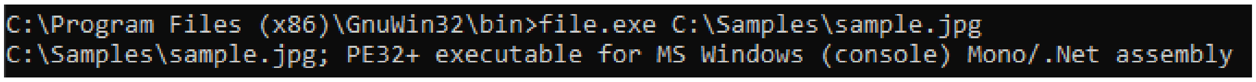
In Dynamic Analysis, we closely and thoroughly observe what happens when we say "LEEEROOOOOY JEEEENKINS*" and double click an armed malware file, in order to investigate the effects and actions of the malware's binary. And while this is not the proper scientific definition, it is the absolute and undeniable truth.

*”Leeroy Jenkins” refers to the viral video of the World of Warcraft player with this name and yelling it indicates bold and daring behavior.

Static Analysis

Identifying File Formats

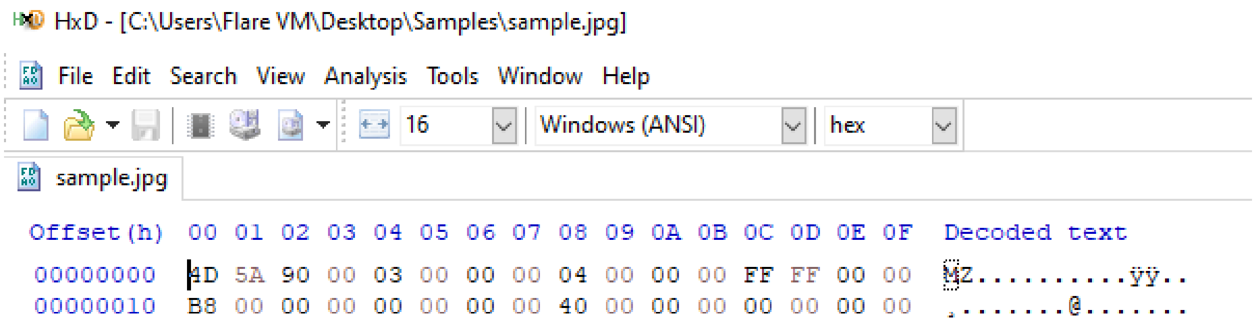
Both in Linux and in Windows systems, only a handful of file formats can be executed. Malicious actors usually cascade the malware in order to appear as something else than what they actually are. This was initially achieved by changing the extension (from .exe to .jpg for example) however there are more sophisticated methods to do so now. Thankfully, there are ways to identify whether a file is executable as well as its original extension in both cases.
The first is a handy little tool named “file” in Linux. A similar application can be downloaded (Last day visited - 1/2/2023) for Windows



```
C:\Program Files (x86)\GnuWin32\bin>file.exe C:\Samples\sample.jpg
C:\Samples\sample.jpg; PE32+ executable for MS Windows (console) Mono/.Net assembly
```

Figure 2.3.1 - Use of the file command

Another, a bit more tricky but definitely sleeker, way to identify a file type, is through the so called “magic numbers”. The magic numbers are the 512 first bytes of a file and are used in order for the identification of a file. The list of magic numbers (last day visited - 31/01/2023) is long and extended, however two types are important to malware analysts. The MZ (named after Microsoft’s developer Mark Zbikowski) files with the magic number “4D 5A” that represent executable Windows files and the ELF files with the magic number “7F 45 4C 46”, representing Unix/Linux executable files.
The identification is by using Hex Editors, tools capable of editing and displaying the bytes of a file. Flare VM has two Hex Editors, HxD and fileinsight.



HxD - [C:\Users\Flare VM\Desktop\Samples\sample.jpg]
File Edit Search View Analysis Tools Window Help
16
Windows (ANSI)
hex
sample.jpg

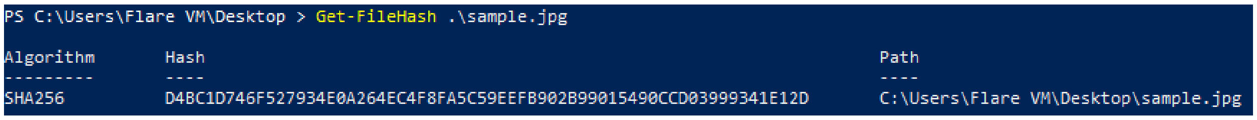
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
```

Figure 2.3.2 - Use of HxD Hex Editor

In both cases, we see that the file sample.jpg is a windows executable file.
More about the Windows executables will be examined later in this chapter, as the Portable Executable file format deserves to be analyzed further and more in depth.

Hashing

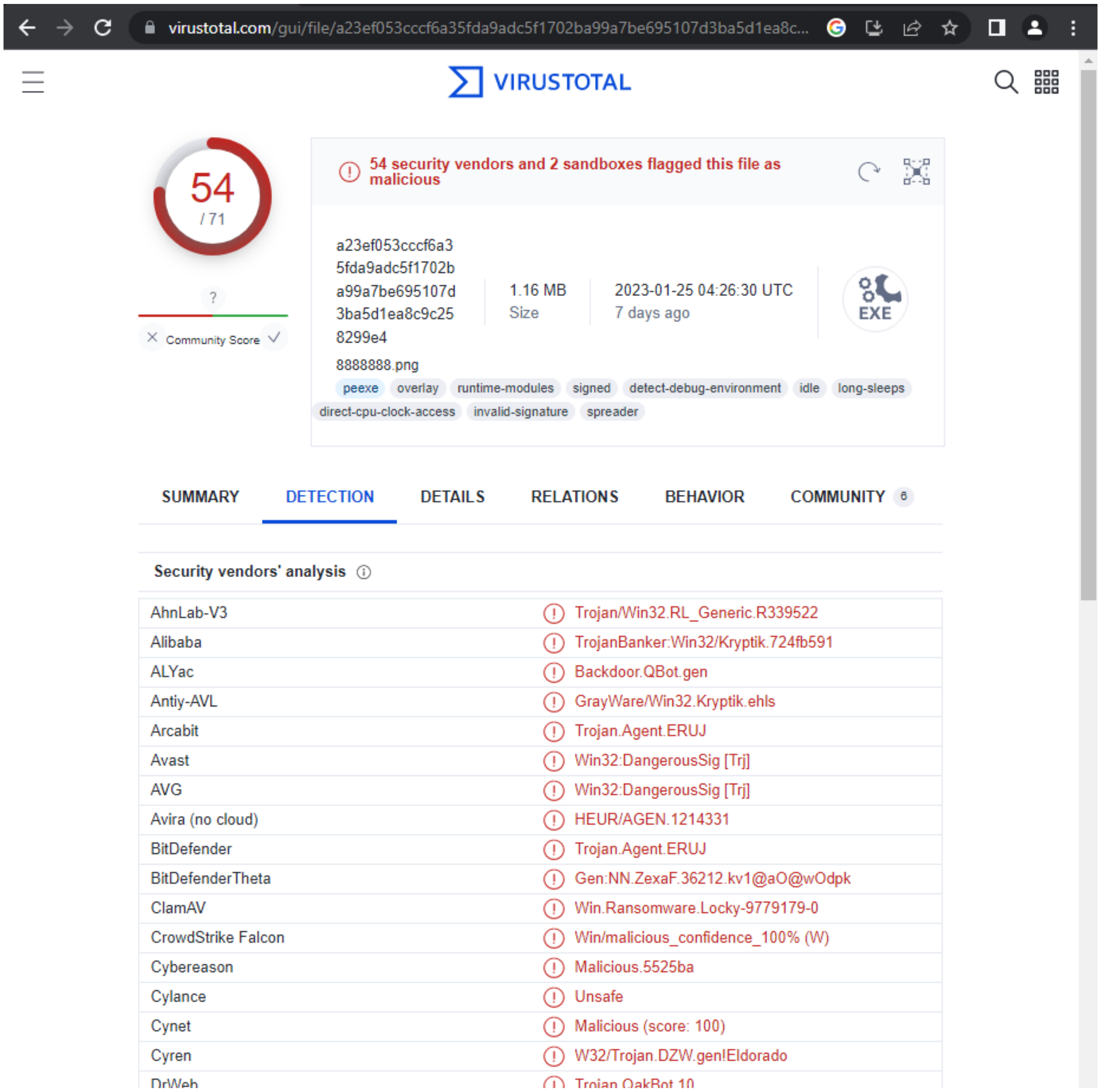
One of the first techniques used in static analysis, especially after identifying a file and realizing it is potentially dangerous, is usually hashing. A hashing algorithm takes as an input a file and delivers a unique checksum. Then by comparing that said output with a database of known malware checksums, the analyst can quickly and safely figure out if the malware is already known and documented. Hashing is the way most anti-virus software used to work and the way they kept the users safe was by constantly updating their database (the procedure has changed on newer anti-malware programs with the utilization of more modern techniques and machine learning models).
There are various hashing algorithms, with the SHA256 and SHA512 being the most prominent ones. MD5 and SHA1 were used in the past but collisions have been found and exploited on them and are no longer considered reliable.
There are various software solutions for Windows, Linux and MacOS users to calculate the hash of a file. Some are cross platform and some have a Graphical User Interface. However, since we, as analysts, want the most efficient way to deal with this, the best way is to use the terminal.
MacOS users can do so with the **shasum** command available on all modern mac computers and Linux users have a ton of options like **RHash**, **sha256sum, MassHash** and **GtkHash**.
On Windows, the **Get-Filehash** Powershell command can provide the checksum utilizing a variety of algorithms, including SHA256 and SHA512. Since Flare is a Windows machine and Windows is the most targeted operating system for malware, we will elaborate a little more on that.



```
PS C:\Users\Flare VM\Desktop > Get-FileHash .\sample.jpg

Algorithm       Hash                                                                    Path
---------       ----                                                                    ----
SHA256          D4BC1D746F527934E0A264EC4F8FA5C59EEFB902B99015490CCD03999341E12D        C:\Users\Flare VM\Desktop\sample.jpg
```

Figure 2.3.3 - Get-Filehash Use and Output

After calculating the hash of a file, we need to actually compare it to known threats. To do that, we can use tools like virustotal.com (last day visited - 30/01/2023), enter the hash and see the results. If we get a match, then the file is a known threat and should be treated accordingly.



virustotal.com/gui/file/a23ef053cccf6a35fda9adc5f1702ba99a7be695107d3ba5d1ea8c...

VIRUSTOTAL

54 / 71

Community Score

54 security vendors and 2 sandboxes flagged this file as malicious

a23ef053cccf6a35fda9adc5f1702ba99a7be695107d3ba5d1ea8c9c258299e4
8888888.png

1.16 MB Size

2023-01-25 04:26:30 UTC 7 days ago

EXE

peexe overlay runtime-modules signed detect-debug-environment idle long-sleeps direct-cpu-clock-access invalid-signature spreader

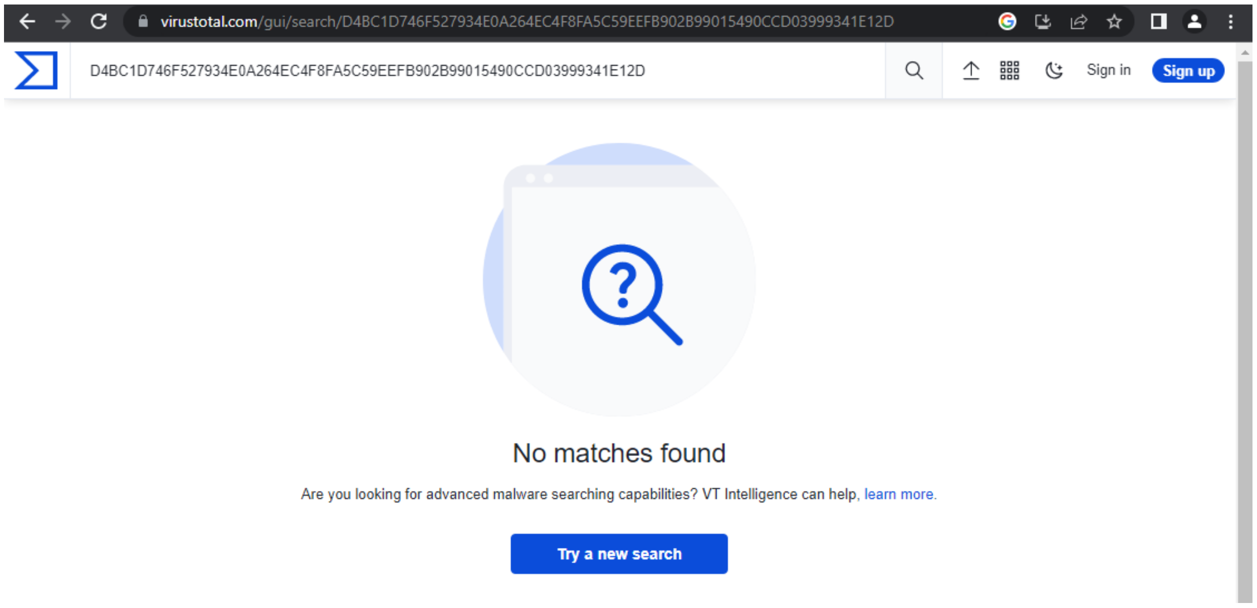
SUMMARY DETECTION DETAILS RELATIONS BEHAVIOR COMMUNITY 6

Security vendors' analysis

AhnLab-V3	Trojan/Win32.RL_Generic.R339522
Alibaba	TrojanBanker:Win32/Kryptik.724fb591
ALYac	Backdoor.QBot.gen
Antiy-AVL	GrayWare/Win32.Kryptik.ehls
Arcabit	Trojan.Agent.ERUJ
Avast	Win32:DangerousSig [Trj]
AVG	Win32:DangerousSig [Trj]
Avira (no cloud)	HEUR/AGEN.1214331
BitDefender	Trojan.Agent.ERUJ
BitDefenderTheta	Gen:NN.ZexaF.36212.kv1@aO@wOdpk
ClamAV	Win.Ransomware.Locky-9779179-0
CrowdStrike Falcon	Win/malicious_confidence_100% (W)
Cybereason	Malicious.5525ba
Cylance	Unsafe
Cynet	Malicious (score: 100)
Cyren	W32/Trojan.DZW.gen!Eldorado
DrWeb	Trojan.QakBot.10

Figure 2.3.4 - virustotal.com output for known threats

Obviously that is a very dangerous file and should not be executed without the necessary precautions. If there was nothing in the various databases virustotal.com has access to, we would have something like the following:


virustotal.com/gui/search/D4BC1D746F527934E0A264EC4F8FA5C59EEFB902B99015490CCD03999341E12D

D4BC1D746F527934E0A264EC4F8FA5C59EEFB902B99015490CCD03999341E12D

No matches found

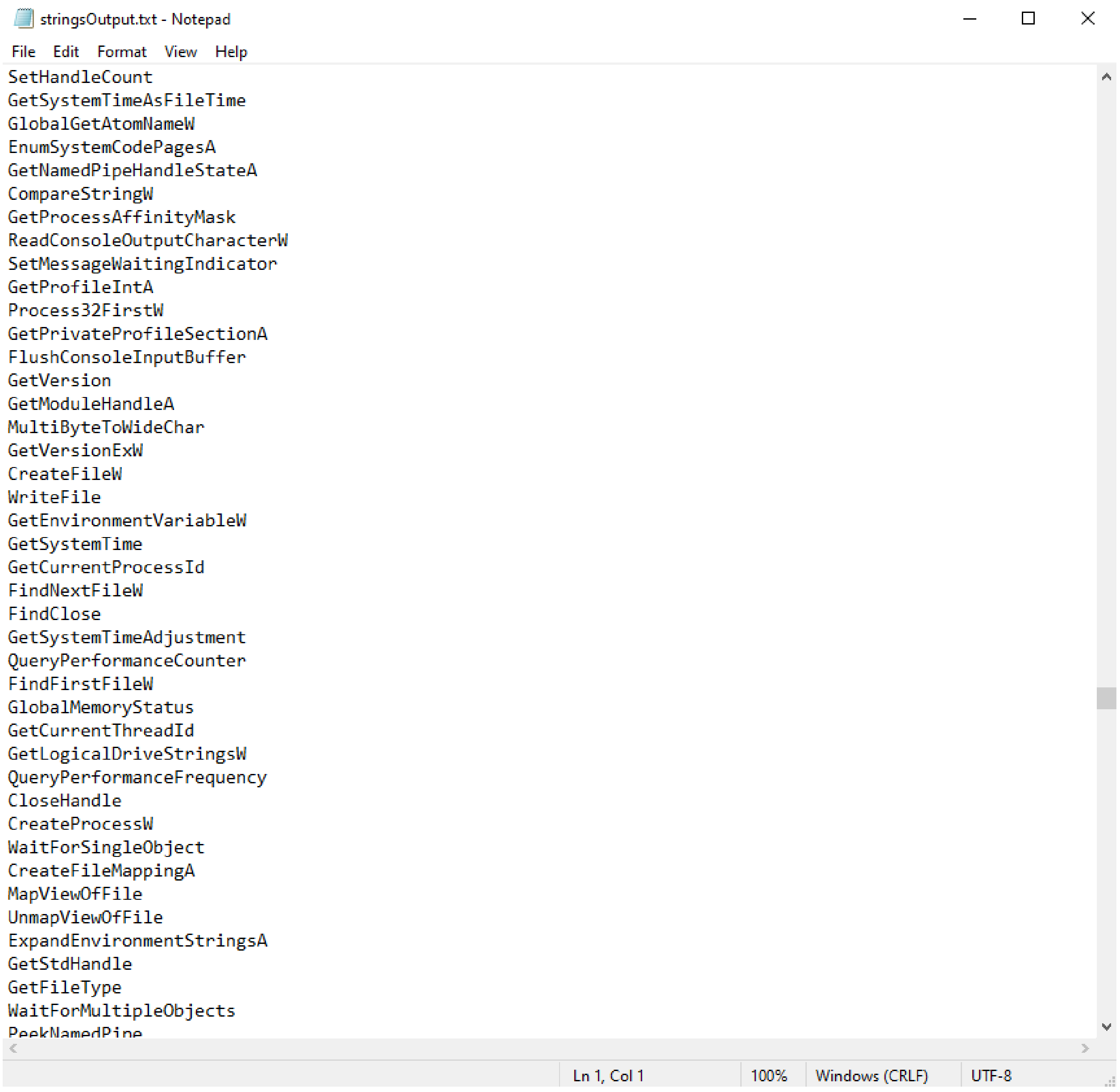
Are you looking for advanced malware searching capabilities? VT Intelligence can help, learn more.

Figure 2.3.5 - virustotal.com output for not matching outcomes

Of course, this entire process is not foolproof. There are three main reasons for it to fail. Either the database is not updated, the malware is a novelty or the malicious actor has spent a decent amount of effort and resources in making sure its hash value is different with each sample, a technique known as hashbusting.
In the first case, we can be relatively sure that virustotal keeps their databases updated. If the malware is a novelty, there is little we can do in Static Analysis, we can however analyze it dynamically, a process we will examine in later chapters.
If the malware is obfuscated, there is little we can do as standard VirusTotal users. A paid Enterprise account can compare hashes and deliver a list of similar malware, the percentage of similarity as well as the number of antivirus software that would detect the malware. Without access to a paid service, we are once again left with dynamic analysis techniques.

Analyzing Strings

After figuring out that a file is indeed executable and malware, we need to see what it would actually do should we run it. One of the techniques we would use for it is string analysis. Strings is a command that works both on Windows and Linux. Also, in both cases, the handy “>” operator can redirect the output to a file for us to see and review. On the same dangerous file we saw earlier, the strings command gives an output worth digging into.



```
SetHandleCount
GetSystemTimeAsFileTime
GlobalGetAtomNameW
EnumSystemCodePagesA
GetNamedPipeHandleStateA
CompareStringW
GetProcessAffinityMask
ReadConsoleOutputCharacterW
SetMessageWaitingIndicator
GetProfileIntA
Process32FirstW
GetPrivateProfileSectionA
FlushConsoleInputBuffer
GetVersion
GetModuleHandleA
MultiByteToWideChar
GetVersionExW
CreateFileW
WriteFile
GetEnvironmentVariableW
GetSystemTime
GetCurrentProcessId
FindNextFileW
FindClose
GetSystemTimeAdjustment
QueryPerformanceCounter
FindFirstFileW
GlobalMemoryStatus
GetCurrentThreadId
GetLogicalDriveStringsW
QueryPerformanceFrequency
CloseHandle
CreateProcessW
WaitForSingleObject
CreateFileMappingA
MapViewOfFile
UnmapViewOfFile
ExpandEnvironmentStringsA
GetStdHandle
GetFileType
WaitForMultipleObjects
PeekNamedPipe
```

Figure 2.3.6 - strings command output

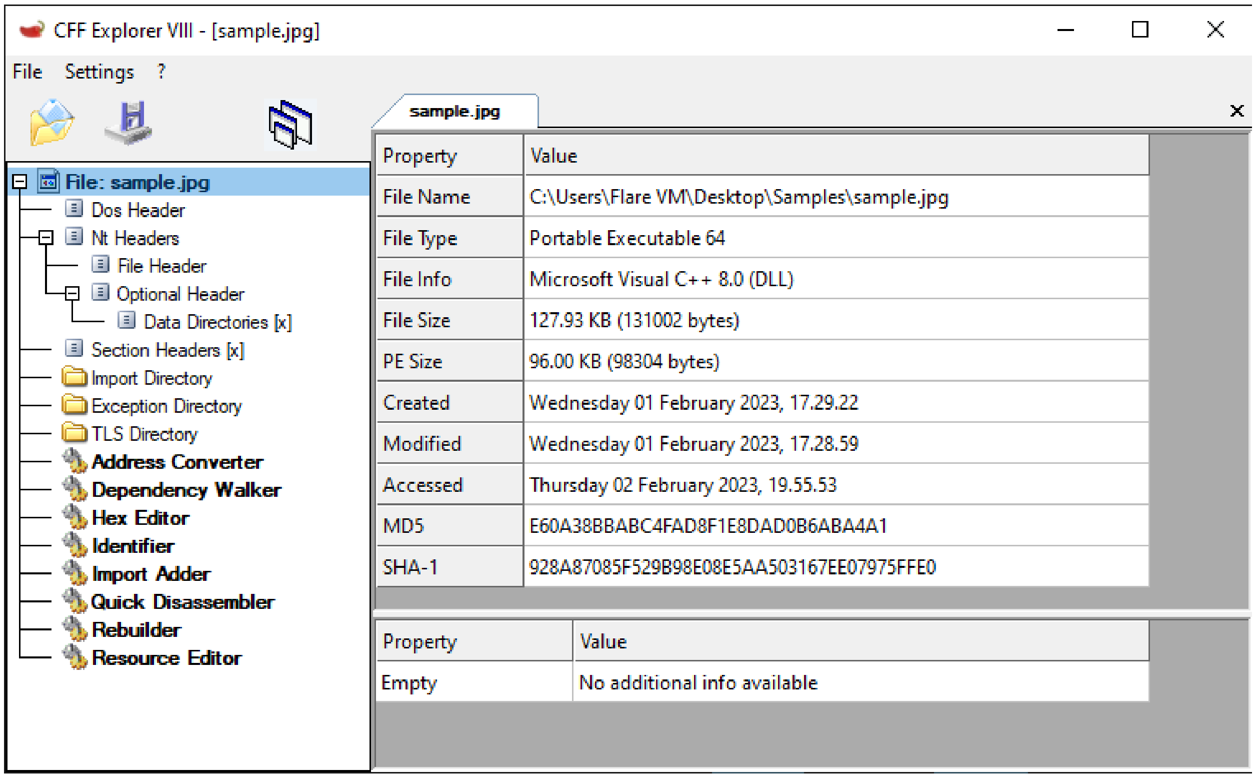
After skipping some nonsensical strings we reach a point where the executable file clearly makes some Windows API calls. Later on, we can see some error handling and at the end of the file, the authors and what this malware was trying to trick the Windows Defender into thinking it was.

The Portable Executable (PE) file format

We earlier saw that all Windows executable files start with the magic numbers “4D 5A”. And while there are various types of executables a Windows machine can run, they all belong to the greater Portable Executable family of files. In this section we will delve in depth to the Portable Executable file and observe what these extensions have in common, from the most obvious .exe (Windows Executable), to the “weird files that all software engineers encounter early in their career” .dll (Dynamic Link Library) and to the incredibly subtle .src (Windows Screensaver), that can be used to run malware on the victim’s system. Namely, a list of these extensions is this:

- Windows Executable (.exe)
- Dynamic Link Library (.dll)
- Shortcut (.lnk)
- Control Panel Item (.cpl)
- Windows Screensaver (.src)
- Drivers (.drv)
- Multilingual User Interface (.mui)
- System (.sys)

Each of these files can be executed in its own way, whether it be as an application of its own, as part of another application or via a legitimate Windows service.
In Flare VM, we can split the executable files and study them individually with a tool named CFF Explorer.



Property	Value
File Name	C:\Users\Flare VM\Desktop\Samples\sample.jpg
File Type	Portable Executable 64
File Info	Microsoft Visual C++ 8.0 (DLL)
File Size	127.93 KB (131002 bytes)
PE Size	96.00 KB (98304 bytes)
Created	Wednesday 01 February 2023, 17.29.22
Modified	Wednesday 01 February 2023, 17.28.59
Accessed	Thursday 02 February 2023, 19.55.53
MD5	E60A38BBABC4FAD8F1E8DAD0B6ABA4A1
SHA-1	928A87085F529B98E08E5AA503167EE07975FFE0

Property	Value
Empty	No additional info available

Figure 2.3.7 - CFF Explorer

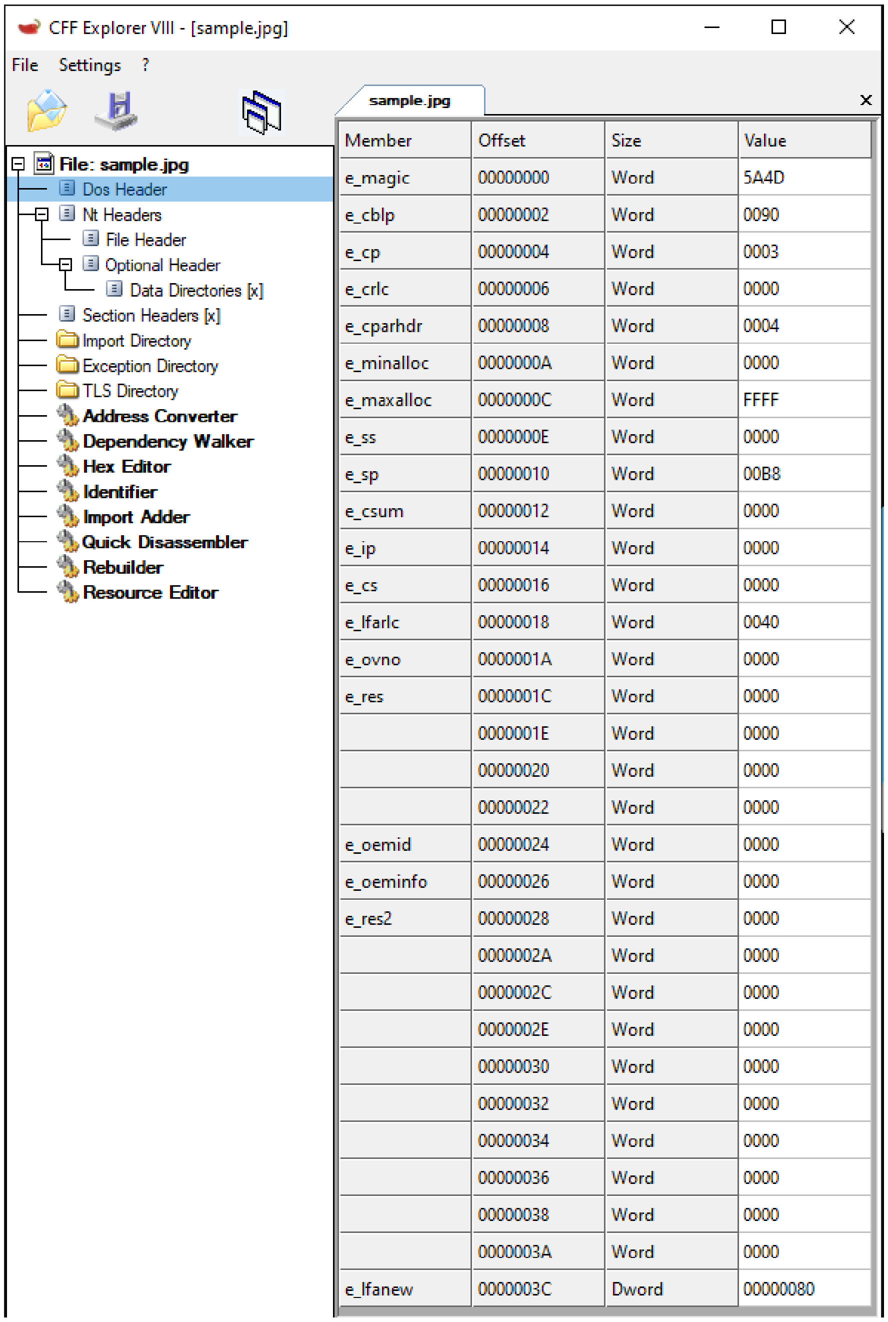
As we see, the file “sample.jpg” is an executable file. We knew that from before, but now we can see the various elements separately.

The PE file format consists of various sections. The first of them are discrete and appear in all files. These sections are:

- DOS Header
- DOS Stub
- PE Header
- Section Table
- Section 1

Then there is an indefinite number of sections, according to each specific file.

The DOS header:
Before Windows became the most widely used operating system for personal computers, Microsoft’s flagship operating system was DOS. And Microsoft spent considerable resources in making every Windows system compatible with the previous ones in order to not lose the market share they had. And thus, almost 30 years after Windows entered the modern household completely replacing and eclipsing DOS, we still have to study the leftover section of the PE file known as the DOS header.



Member	Offset	Size	Value
e_magic	00000000	Word	5A4D
e_cblp	00000002	Word	0090
e_cp	00000004	Word	0003
e_crlc	00000006	Word	0000
e_cparhdr	00000008	Word	0004
e_minalloc	0000000A	Word	0000
e_maxalloc	0000000C	Word	FFFF
e_ss	0000000E	Word	0000
e_sp	00000010	Word	00B8
e_csum	00000012	Word	0000
e_ip	00000014	Word	0000
e_cs	00000016	Word	0000
e_lfarlc	00000018	Word	0040
e_ovno	0000001A	Word	0000
e_res	0000001C	Word	0000
	0000001E	Word	0000
	00000020	Word	0000
	00000022	Word	0000
e_oemid	00000024	Word	0000
e_oeminfo	00000026	Word	0000
e_res2	00000028	Word	0000
	0000002A	Word	0000
	0000002C	Word	0000
	0000002E	Word	0000
	00000030	Word	0000
	00000032	Word	0000
	00000034	Word	0000
	00000036	Word	0000
	00000038	Word	0000
	0000003A	Word	0000
e_lfanew	0000003C	Dword	00000080

Figure 2.3.8 - The DOS header

The two members we are mostly interested in are the “e_magic” and the “e_ifanew”. e_magic is the magic numbers we described earlier. e_ifanew is the last 4 bytes and point to the PE Header (effectively skipping the DOS Stub section)

The DOS Stub

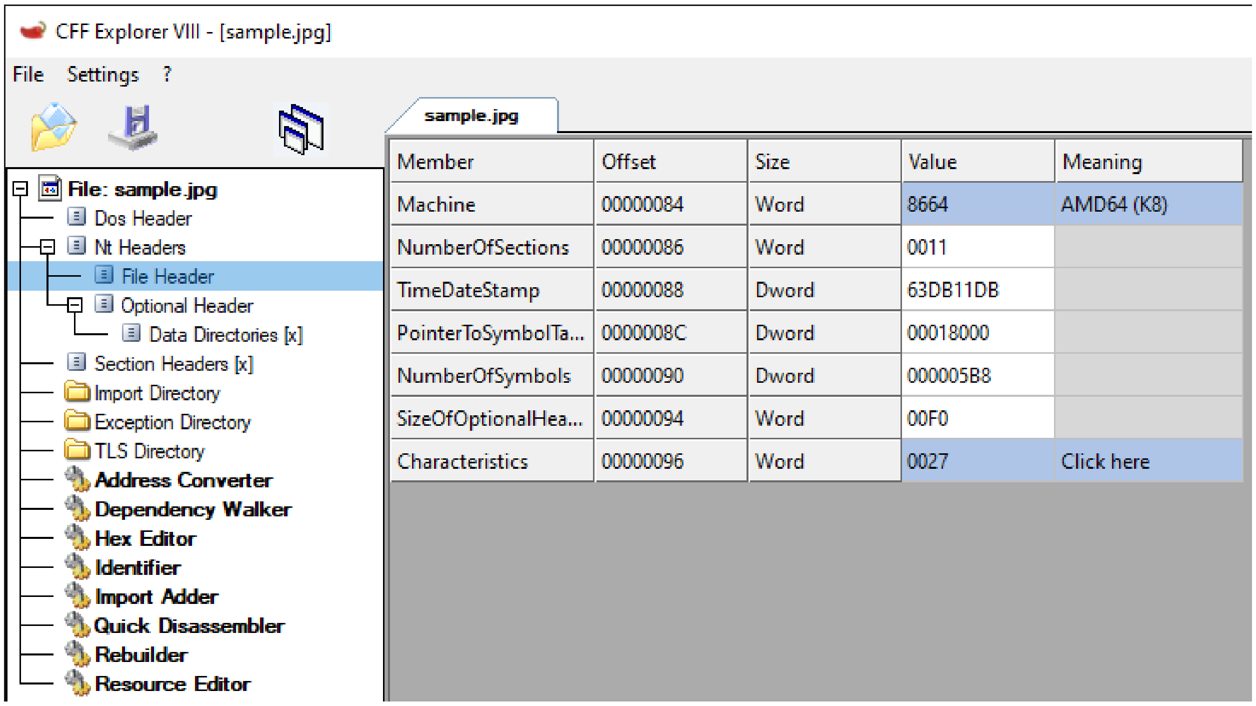
Just like the DOS header, the DOS stub also remained as a means for backwards compatibility. It usually simply contains the line: “This program cannot be run in DOS mode”. Opening a file with HxD, we can see it.

```
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
```

Figure 2.3.9 - The DOS stub

The PE Header

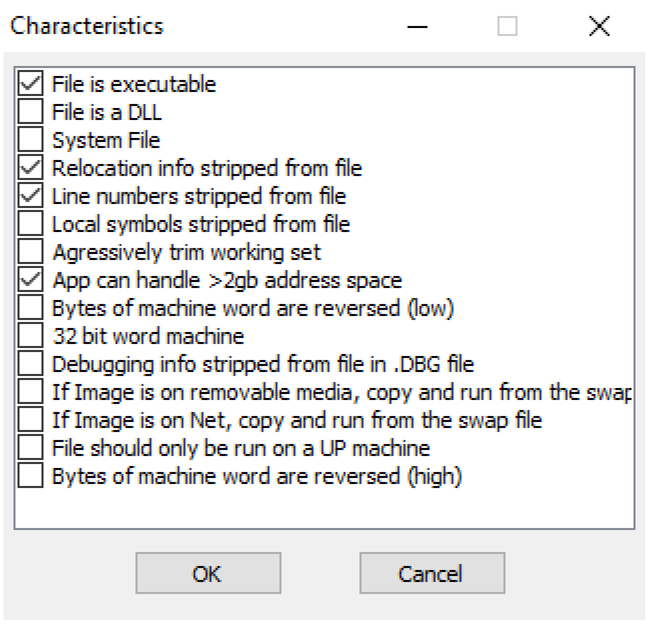
The PE header (also known as Nt header from the NT (New Technology) windows system) starts after the offset indicated at e_ifanew. The PE is very useful for static analysis as it provides information like the architecture the executable is built on, the number of sections (which is fluid as stated earlier) and the characteristics tab that can be used for even more reconnaissance.



Member	Offset	Size	Value	Meaning
Machine	00000084	Word	8664	AMD64 (K8)
NumberOfSections	00000086	Word	0011	
TimeDateStamp	00000088	Dword	63DB11DB	
PointerToSymbolTa...	0000008C	Dword	00018000	
NumberOfSymbols	00000090	Dword	000005B8	
SizeOfOptionalHea...	00000094	Word	00F0	
Characteristics	00000096	Word	0027	Click here

Figure 2.3.10 - The PE header

The Characteristics popup gives additional information. Some, like the architecture (“32 bit word machine” field) or the file type we already knew, but it is worth it for the analyst to spend a few seconds checking it.


Characteristics

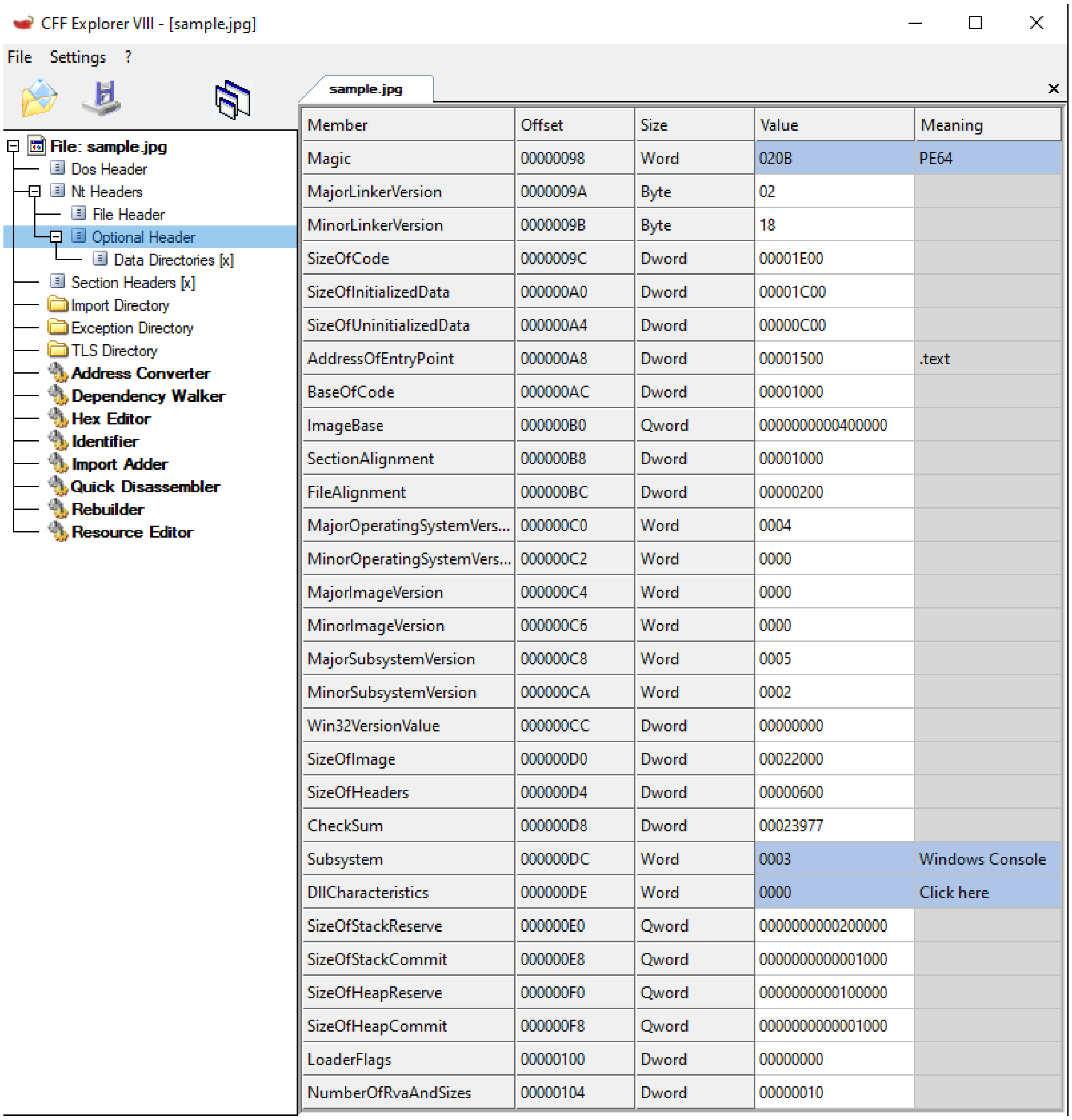
- [✓] File is executable
- [ ] File is a DLL
- [ ] System File
- [✓] Relocation info stripped from file
- [✓] Line numbers stripped from file
- [ ] Local symbols stripped from file
- [ ] Agressively trim working set
- [✓] App can handle >2gb address space
- [ ] Bytes of machine word are reversed (low)
- [ ] 32 bit word machine
- [ ] Debugging info stripped from file in .DBG file
- [ ] If Image is on removable media, copy and run from the swap
- [ ] If Image is on Net, copy and run from the swap file
- [ ] File should only be run on a UP machine
- [ ] Bytes of machine word are reversed (high)

OK Cancel

Figure 2.3.11 - The Characteristics popup

The optional header contains even more information, most importantly metadata that can be both interesting and useful to at least view. Each field has its purpose, however what we are mostly interested in are the following:

- Magic: This field represents whether the executable is 64-bit (with value: 020B) or 32-bit (with value: 010B).
- Address Of Entry Point: This represents the memory address that the code begins.
- Major Operating System Version: This represents the minimum version of the operating system required for the file to be executed. This specific file used in the figures (which is actually a simple Hello World written in C) can be used in all major Windows Systems.
- Subsystem: This value represents whether the executable has a graphical user interface (GUI) or runs in the console (CLI).
- DLL Characteristics: If the is a DLL, there is crucial information like the program's ability to understand if it is running on a terminal, a server, as a service and most importantly whether or not it has the ability to move within the system's memory.



Member	Offset	Size	Value	Meaning
Magic	00000098	Word	020B	PE64
MajorLinkerVersion	0000009A	Byte	02	
MinorLinkerVersion	0000009B	Byte	18	
SizeOfCode	0000009C	Dword	00001E00	
SizeOfInitializedData	000000A0	Dword	00001C00	
SizeOfUninitializedData	000000A4	Dword	00000C00	
AddressOfEntryPoint	000000A8	Dword	00001500	.text
BaseOfCode	000000AC	Dword	00001000	
ImageBase	000000B0	Qword	0000000000400000	
SectionAlignment	000000B8	Dword	00001000	
FileAlignment	000000BC	Dword	00000200	
MajorOperatingSystemVers...	000000C0	Word	0004	
MinorOperatingSystemVers...	000000C2	Word	0000	
MajorImageVersion	000000C4	Word	0000	
MinorImageVersion	000000C6	Word	0000	
MajorSubsystemVersion	000000C8	Word	0005	
MinorSubsystemVersion	000000CA	Word	0002	
Win32VersionValue	000000CC	Dword	00000000	
SizeOfImage	000000D0	Dword	00022000	
SizeOfHeaders	000000D4	Dword	00000600	
CheckSum	000000D8	Dword	00023977	
Subsystem	000000DC	Word	0003	Windows Console
DllCharacteristics	000000DE	Word	0000	Click here
SizeOfStackReserve	000000E0	Qword	0000000000200000	
SizeOfStackCommit	000000E8	Qword	0000000000001000	
SizeOfHeapReserve	000000F0	Qword	0000000000100000	
SizeOfHeapCommit	000000F8	Qword	0000000000001000	
LoaderFlags	00000100	Dword	00000000	
NumberOfRvaAndSizes	00000104	Dword	00000010	

Figure 2.3.12 - The Optional Header tab

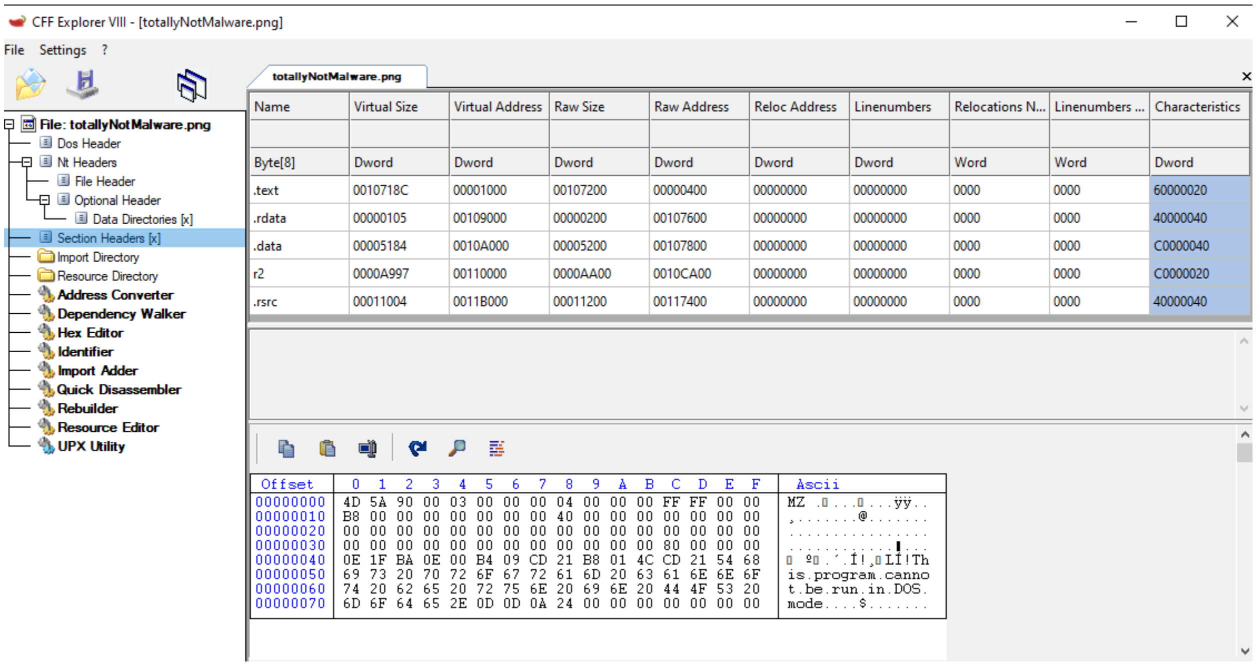
<u>The Section Header</u>

As discussed before there can be an indefinite number of sections in the PE file format. The Section Header can be a valuable resource because it mostly consists of the same parts. An experienced analyst can identify sections that indicate an attempt to obfuscate code.

The standard parts are:

- .text: contains the code.

- .rdata: contains data that is read-only, for example strings
- .data: contains mutable data
- .rsrc: contains various resources like images, sounds, icons and other assets.
- .idata: contains the Imports Address Table, that we will see soon.



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0010718C	00001000	00107200	00000400	00000000	00000000	0000	0000	60000020
.rdata	00000105	00109000	00000200	00107600	00000000	00000000	0000	0000	40000040
.data	00005184	0010A000	00005200	00107800	00000000	00000000	0000	0000	C0000040
r2	0000A997	00110000	0000AA00	0010CA00	00000000	00000000	0000	0000	C0000020
.rsrc	00011004	0011B000	00011200	00117400	00000000	00000000	0000	0000	40000040

```
Offset     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    Ascii
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00   MZ ..........ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ,.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00   ............▮...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68   . º..´.Í!¸.LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F   is.program.canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20   t.be.run.in.DOS.
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00   mode....$.......
```

Figure 2.3.13 - The Section Table

The section “r2” is definitely not standard. Maybe a malicious actor used some sort of packer to bypass outdated antivirus software.

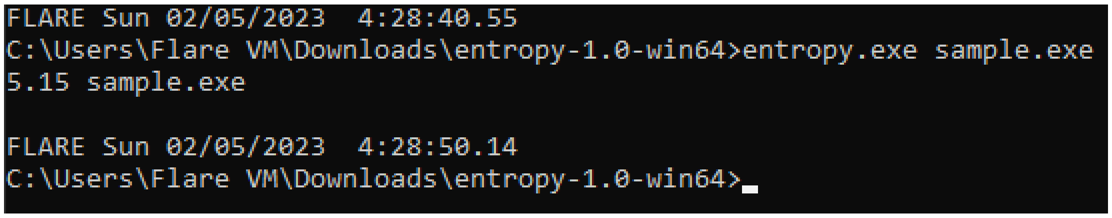
The Import Access Table

The Import Access Table, also referred to as Import Directory, allows us to inspect the number of functions that get imported. As we saw earlier, at the Analyzing Strings section, malware often make API calls and take advantage of built-in Windows features. On the IAT there is a list of all these calls, classified under the correspondent dll. Then, simply visiting Microsoft’s official documentation, we can get a rough idea of the data this program collects and what it does with it. Of course, these calls are not nefarious by themselves, but some of them (HTTP requests, file enumeration, keystroke control, socket opening just to name a few) should raise serious alarms. An excellent resource for these calls is ‘A Novel Approach to Detect Malware Based on API Call Sequence Analysis’ by Youngjoon Ki, Eunjin Kim, and Huy Kang Kim [3] (https://journals.sagepub.com/doi/full/10.1155/2015/659101 last day visited: 14/2/2023).

Identifying Packers

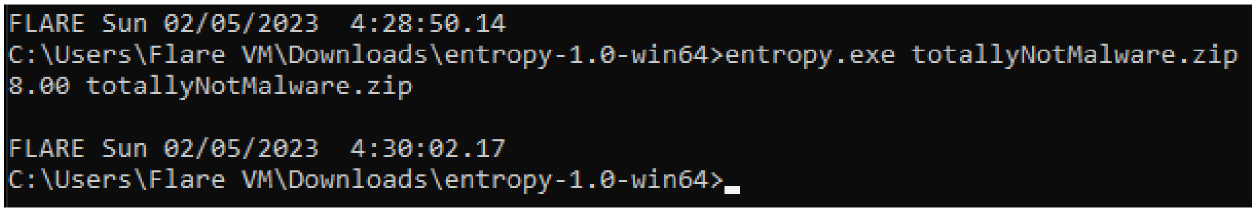
The main functionality of Packers is to compress and potentially encrypt data or files into a single file. However a more sinister application is to hide an executable before it gets unpacked. Luckily, there are tools and techniques that can help with identifying a packed file. One of these methods we have already seen in the previous section, where examining the Section Table and the Import Access Table can provide enough intel to understand if this is the case.

Another way to figure out if a packer was used has to do with entropy. By definition, entropy is the randomness and the lack of predictability of the bytes of a file. A high amount of entropy usually indicates that a packer has been used. Flare VM does not come with a tool to calculate the entropy of a file, however there is a handy little tool simply named “entropy” (last day visited - 5/2/2023) that can assist us.



```
FLARE Sun 02/05/2023  4:28:40.55
C:\Users\Flare VM\Downloads\entropy-1.0-win64>entropy.exe sample.exe
5.15 sample.exe

FLARE Sun 02/05/2023  4:28:50.14
C:\Users\Flare VM\Downloads\entropy-1.0-win64>
```

Figure 2.3.14 - Low Entropy file



```
FLARE Sun 02/05/2023  4:28:50.14
C:\Users\Flare VM\Downloads\entropy-1.0-win64>entropy.exe totallyNotMalware.zip
8.00 totallyNotMalware.zip

FLARE Sun 02/05/2023  4:30:02.17
C:\Users\Flare VM\Downloads\entropy-1.0-win64>
```

Figure 2.3.15 - High Entropy File

According to the documentation, values greater than 7 indicate the use of a compression algorithm.

Dynamic Analysis

Theoretical Background

There are a few major questions when it comes to dynamic analysis.

1. What does the malware do now?
2. Does it create any types of persistence in the victim's machine?
3. Does it remain dormant and if yes, for how long?
4. Does it try to get access to the rest of the network and if yes, in what way?
5. Does it try to get elevated privileges?

Some of these questions must be answered in real time, as the malware runs while others can be answered at a later time. In any cases, dynamic analysis is more fluid and demanding and we often use techniques taken from the static analysis handbook.

Identifying Processes

Since we will be purposefully executing malware, we first need to understand that files can remain dormant and harmless in a system. It is a process that we, as malware analysts, should be careful of. Both in Windows and in Linux systems, the file, once executed, is copied to the system's memory (either RAM or Virtual Memory/Swap) and then it gets executed by the operating system. So, now that we know what a process is, we should see how to actually monitor them. In Linux systems there are obviously numerous options to do so, with the "top" command being preinstalled and the "htop" command that builds on top and delivers information in a more coherent way. In Windows systems, the Task Manager can be a useful tool, however Flare VM offers "Process Hacker" as a better alternative. In both cases, the process has an id (PID) and we can get the file that created the process as well as the resources it consumes.
These tools are the first defense against cryptojacking malware, since they can sort the processes according to the resources allocated to them.

Analyzing Packets

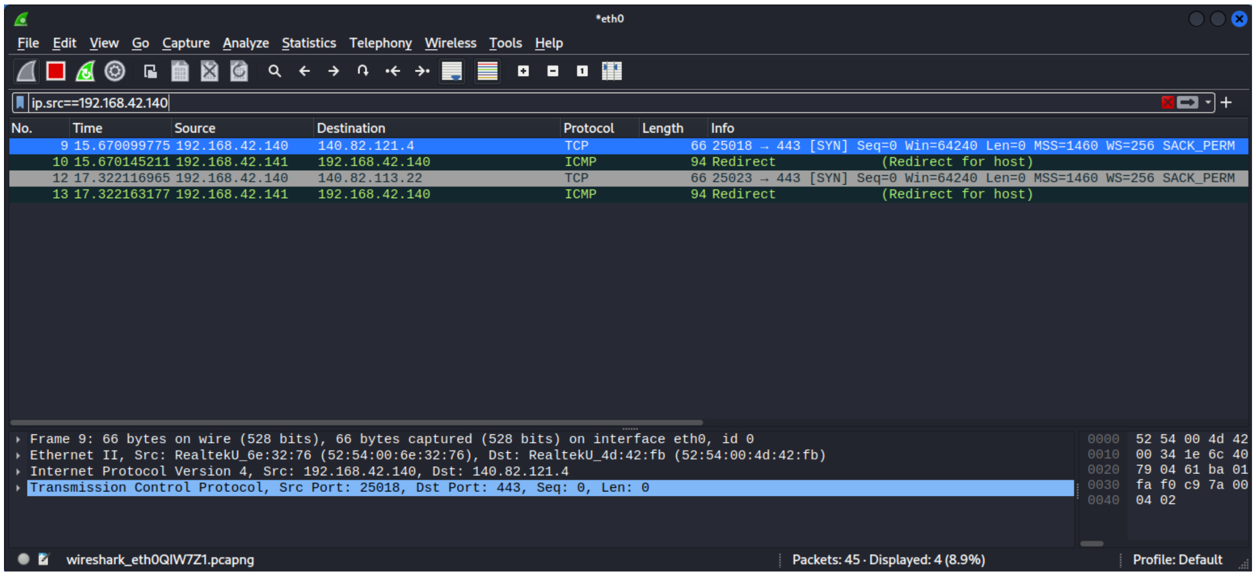
In order to analyze the packets we use Wireshark on the Kali VM and set the filter

```
ip.src == 192.168.42.140
```

to get all traffic originating from Flare VM. If we have blocked Flare's access to the internet, as the filter we should use

```
ip.src == 192.168.42.140 and udp.port != 53
```

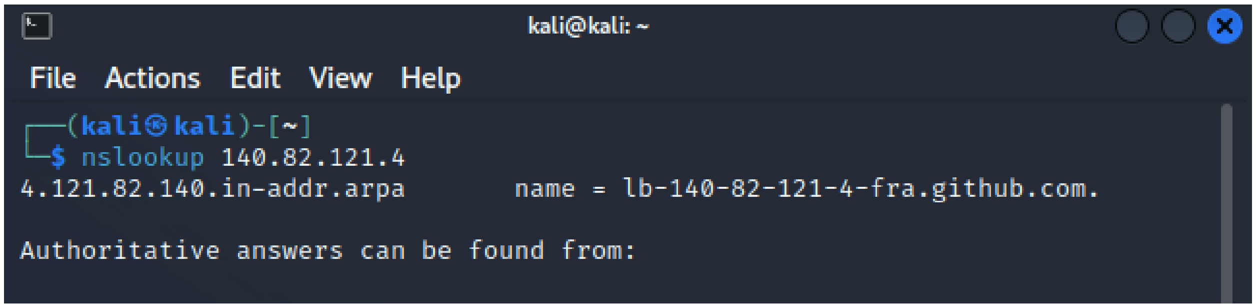
to drop dns requests that tend to overwhelm the workspace.


*eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
ip.src==192.168.42.140

No.  Time         Source          Destination     Protocol  Length  Info
   9 15.670099775 192.168.42.140  140.82.121.4    TCP       66 25018 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
  10 15.670145211 192.168.42.141  192.168.42.140  ICMP      94 Redirect             (Redirect for host)
  12 17.322116965 192.168.42.140  140.82.113.22   TCP       66 25023 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
  13 17.322163177 192.168.42.141  192.168.42.140  ICMP      94 Redirect             (Redirect for host)

› Frame 9: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface eth0, id 0      0000  52 54 00 4d 42
› Ethernet II, Src: RealtekU_6e:32:76 (52:54:00:6e:32:76), Dst: RealtekU_4d:42:fb (52:54:00:4d:42:fb)  0010  00 34 1e 6c 40
› Internet Protocol Version 4, Src: 192.168.42.140, Dst: 140.82.121.4                              0020  79 04 61 ba 01
› Transmission Control Protocol, Src Port: 25018, Dst Port: 443, Seq: 0, Len: 0                   0030  fa f0 c9 7a 00
                                                                                                  0040  04 02

wireshark_eth0QIW7Z1.pcapng          Packets: 45 · Displayed: 4 (8.9%)          Profile: Default
```

Figure 2.4.1 - Wireshark

The destination field is the important one since most modern malware will try to contact a Command and Control server. This server will probably be a computer that the hackers have access to and route their traffic through it. Once we have this information, we can use the nslookup command to find the domain the IP corresponds to. In this case it is a legitimate github page (and it is accurate since I just refreshed a browser tab for demonstration purposes), but a proper malware will try to connect to a weirder address.



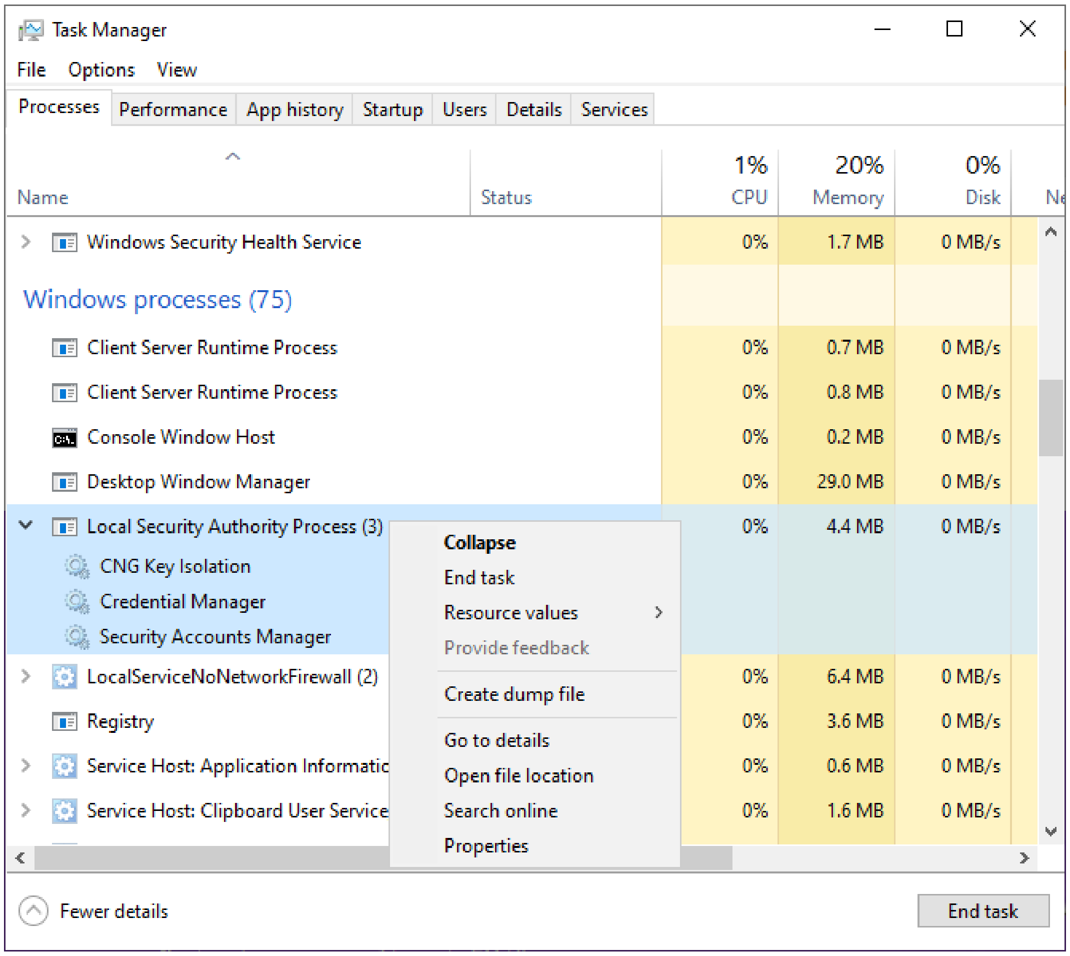
```
┌──(kali㉿kali)-[~]
└─$ nslookup 140.82.121.4
4.121.82.140.in-addr.arpa       name = lb-140-82-121-4-fra.github.com.

Authoritative answers can be found from:
```

Figure 2.4.2 - nslookup

Privilege Escalation

There are two major ways for the hacker to get administrator rights. The first and most common is the use of Mimikatz[4], a tool that was ironically developed to display the security flaws of Windows security services but ended up being misused on a standard basis from hackers to achieve privilege escalation. And while the stock Windows

Defender has become good in detecting and stopping Mimikatz specifically, the overall security flaws still exist. And they are utilized without raising the alarms like Mimikatz does.
In the following figure, we display how every user, malicious or not, can create a file containing NTLM hashes. NTLM is the hashing algorithm used for passwords on a Windows system and has been found to have critical vulnerabilities. By selecting "Create dump file" a user can get that file and exploit one of these vulnerabilities to get all passwords, even administrative ones. Note that I said "user". All users have access to this feature, legitimate or not.

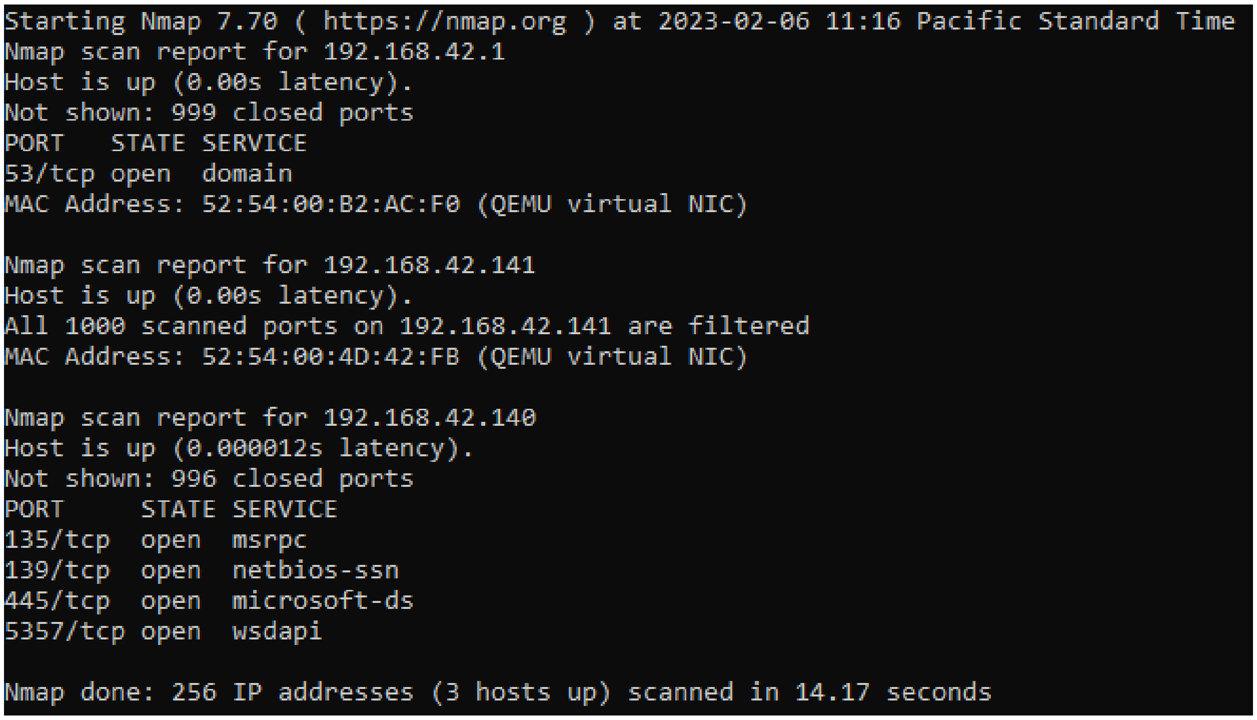


Name	Status	1% CPU	20% Memory	0% Disk
Windows Security Health Service		0%	1.7 MB	0 MB/s
Windows processes (75)				
Client Server Runtime Process		0%	0.7 MB	0 MB/s
Client Server Runtime Process		0%	0.8 MB	0 MB/s
Console Window Host		0%	0.2 MB	0 MB/s
Desktop Window Manager		0%	29.0 MB	0 MB/s
Local Security Authority Process (3)		0%	4.4 MB	0 MB/s
CNG Key Isolation				
Credential Manager				
Security Accounts Manager				
LocalServiceNoNetworkFirewall (2)		0%	6.4 MB	0 MB/s
Registry		0%	3.6 MB	0 MB/s
Service Host: Application Informatic		0%	0.6 MB	0 MB/s
Service Host: Clipboard User Service		0%	1.6 MB	0 MB/s

Figure 2.4.3 - The Local Security Authority Subsystem Process (LSASS.exe)

Network Enumeration

Once an attacker has a hold of a system and has administrator privileges, they can use a wide array of networking tools to scan the rest of the network and decide whether or not it is worth their time to exploit it further. Note that these tools are, once again, legitimate and are used by network engineers to setup, control and test the communication capabilities of a network. There are various tools for this with the most common one being nmap, a tool present in Windows (and most Linux) systems. Nmap is a Command Line Interface tool and most hackers learn about it early on in their career.



```
Starting Nmap 7.70 ( https://nmap.org ) at 2023-02-06 11:16 Pacific Standard Time
Nmap scan report for 192.168.42.1
Host is up (0.00s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
53/tcp open  domain
MAC Address: 52:54:00:B2:AC:F0 (QEMU virtual NIC)

Nmap scan report for 192.168.42.141
Host is up (0.00s latency).
All 1000 scanned ports on 192.168.42.141 are filtered
MAC Address: 52:54:00:4D:42:FB (QEMU virtual NIC)

Nmap scan report for 192.168.42.140
Host is up (0.000012s latency).
Not shown: 996 closed ports
PORT     STATE SERVICE
135/tcp  open  msrpc
139/tcp  open  netbios-ssn
445/tcp  open  microsoft-ds
5357/tcp open  wsdapi

Nmap done: 256 IP addresses (3 hosts up) scanned in 14.17 seconds
```

Figure 2.4.4 - Nmap

This is not a network worth attacking further, however in most organizations, the machines communicate with each other and usually contain the same vulnerabilities. Also, if there is an Active Directory setup, the domain controller becomes an immediate target since they possess the credentials for all the users.

Persistence

Most malware employ some form of persistence in the victim's machine. In Windows machines there are several ways to achieve this. Our job is to identify and eradicate this persistence. Before delving deeper in the persistence techniques and tools, it is worth noting that sometimes malware will simply exist at the folder:

C:\Users\$username\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup

Finding anything weird there should be promptly followed by an investigation and the quarantine of the said application.

Task Manager

The trusty Task Manager, while not foolproof, can once again provide useful information. Heading to the “Startup” tab, we will be greeted with a list of applications that have the capability to run when the computer starts. There, we can disable any malicious application from starting and by right-clicking and selecting “Open File Location” from the menu, we can view the file that gets executed on startup.
While a good support tool however, it is crucial to understand that we cannot entirely rely on the Task Manager to provide all the information we might need. That is because there are services that can bypass our choice. And while we can also view them under the “Services” tab, the sheer number of legitimate services running can easily drown and hide a malicious one.

Registry

By opening the registry editor app (simply searching “regedit” on the start menu will display it), we get better access to the entire system. Here, we must look for the Run and RunOnce keys, existing under:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
```

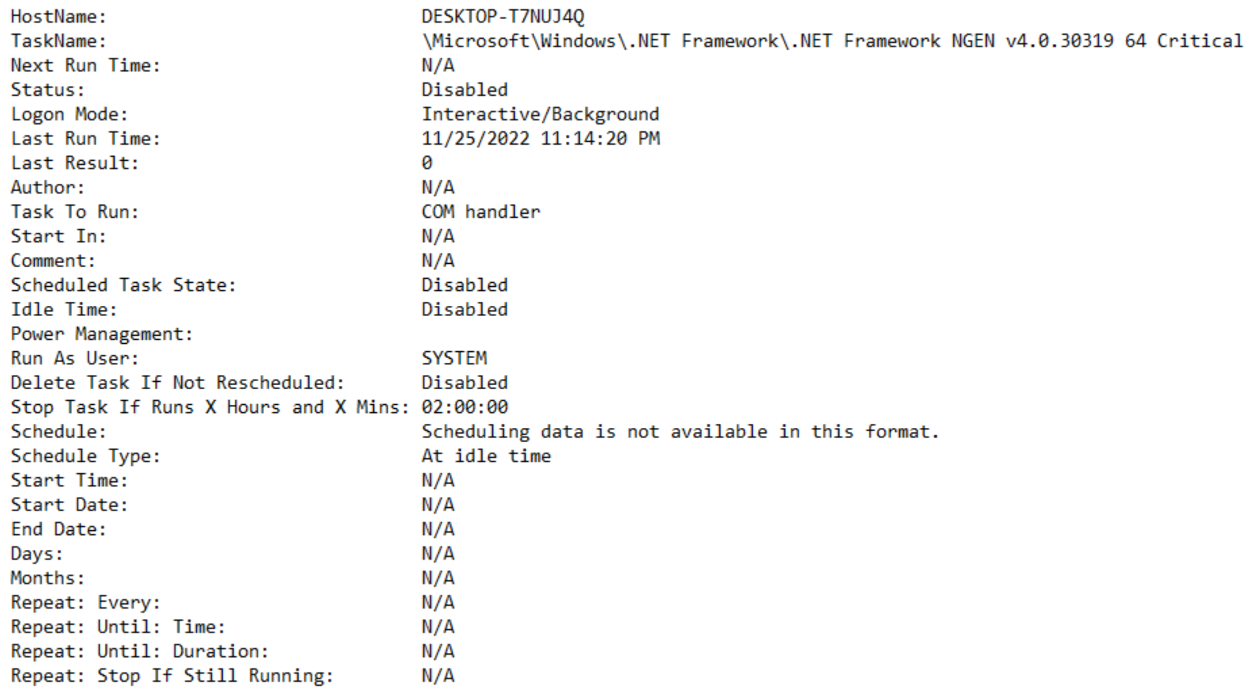
The first thing we can notice here is whether or not the hacker got administrative privileges, since everything under HKEY_LOCAL_MACHINE is system wide and everything under HKEY_CURRENT_USER is specifically for the active user.
It is worth noting that most malware will have a Run and NOT a RunOnce key, since they want to be executed every time the computer starts. Malware with the RunOnce key, usually contain another form of persistence after they get executed.

Task Scheduler

The Task Scheduler is another legitimate tool that can be used for nefarious purposes. Its biggest strength when it comes to malware, is its ability to keep a file dormant and thus, almost invisible until a certain moment in time, whether that is a specific date or after a set amount of time has passed. This is important because most automated malware analysis solutions scan for the first minutes the computer is on and then happily stop consuming resources. One additional way the Task Scheduler benefits malware is by running the malware again after a set amount of time if the process crushes or gets disrupted by the user.
We, as analysts, can view the scheduled tasks with the command

```
schtasks /query /fo list /v > scheduledTasks.txt
```

This will be a long list that far exceeds the Command Prompt's default line limit and that is why it is better for us to redirect the output to a simple text file and review it at our leisure. On the following figure we can see everything we can extract from each record. Each field is pretty much self-explanatory and the amount of information present is vital for our analysis.



```
HostName:                             DESKTOP-T7NUJ4Q
TaskName:                             \Microsoft\Windows\.NET Framework\.NET Framework NGEN v4.0.30319 64 Critical
Next Run Time:                        N/A
Status:                               Disabled
Logon Mode:                           Interactive/Background
Last Run Time:                        11/25/2022 11:14:20 PM
Last Result:                          0
Author:                               N/A
Task To Run:                          COM handler
Start In:                             N/A
Comment:                              N/A
Scheduled Task State:                 Disabled
Idle Time:                            Disabled
Power Management:
Run As User:                          SYSTEM
Delete Task If Not Rescheduled:       Disabled
Stop Task If Runs X Hours and X Mins: 02:00:00
Schedule:                             Scheduling data is not available in this format.
Schedule Type:                        At idle time
Start Time:                           N/A
Start Date:                           N/A
End Date:                             N/A
Days:                                 N/A
Months:                               N/A
Repeat: Every:                        N/A
Repeat: Until: Time:                  N/A
Repeat: Until: Duration:              N/A
Repeat: Stop If Still Running:        N/A
```

Figure 2.4.5 - Task Scheduler record

<u>Shortcuts</u>

The final technique is the masquerading of shortcuts that will, once clicked by the user, first run the malware and then run the software the user intended to. It is by far the least elegant technique and at the same time one of the more deceptive ones for the same reason: It requires user interaction.

It is not elegant because none can be sure that the user will actually press on the shortcut. Even if the software is vital for their day to day life, there are always more ways to execute a program without relying on its shortcut.

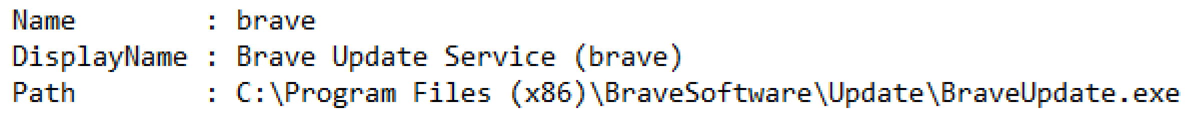
It is deceptive because there is nothing system-wise for the analyst to figure out. Nothing on startup, nothing on the registry, no weird active services and nothing on the task scheduler. The malware runs seemingly at random and without clear indication as to why that happens.

<u>Services</u>

Services, one of the most common techniques used due to their reliability. As we saw earlier, the Task Manager has a separate tab for services. It is cluttered however with a ton of legitimate services that Windows use to operate. Just like with the Task Scheduler, there is a powershell command capable of finding all the services and outputting them in a file:

```
Get-WmiObject win32_service | select Name, DisplayName, @{Name='Path'; Expression={$_.PathName.split('"')[1]}} | Format-List > servicesOutput.txt
```

An example of these records is the following figure:



```
Name        : brave
DisplayName : Brave Update Service (brave)
Path        : C:\Program Files (x86)\BraveSoftware\Update\BraveUpdate.exe
```

Figure 2.4.6 - Service record

While considerably shorter, this simple line provides enough information to locate and purge unwanted services.

Understanding Subterfuge Techniques

Malware does not want to be discovered. And malware authors can go to extreme lengths in order to avoid detection by tools and analysts. The most basic technique used is the avoidance of certain tools. To do so, malware scans the processes and if at any point finds a process named under a blacklisted application, the malware either suspends or terminates itself, until the next time it starts again under its persistence rules that we described earlier. One of the most common ways for a malware analyst to bypass this feature is to simply change the name of the tool they are using. Advanced analysts can develop custom tools, but that is an unnecessary reinvention of the wheel.
Another, way more advanced, way for malware to avoid detection is process injection. According to Ashkan Hosseini[5] (https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process last day visited: 14/2/2023), process injection is “a widespread defense evasion technique employed often within malware and fileless adversary tradecraft, and entails running custom code within the address space of another process”. It is achieved by the use of API calls capable of creating processes, allocating memory, copying data to the said memory, creating threads and in general, performing all variations of nasty stuff.

Classic DLL Injection

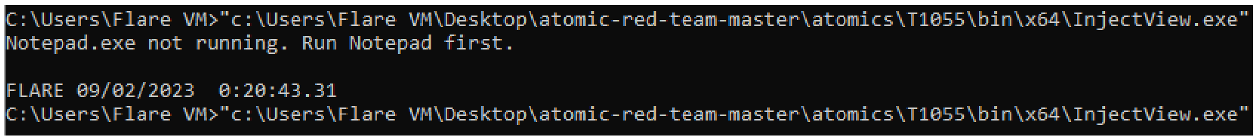
The first process injection technique starts by enumerating the running processes. The malware searches for a process that it can utilize and that has the desirable privilege rights. Once it has a target, it will use the *VirtualAlloc* API call to extend its memory before using *WriteProcessMemory* to add the path to a specific dll. Then, it will simply run *CreateRemoteThread* giving the malicious library enough resources to get loaded and executed.

Portable Executable Injection
The only difference PE Injection has with the Classic DLL Injection is the fact that instead of loading a dll to the memory, it writes a completely new Portable Executable file and runs it directly.

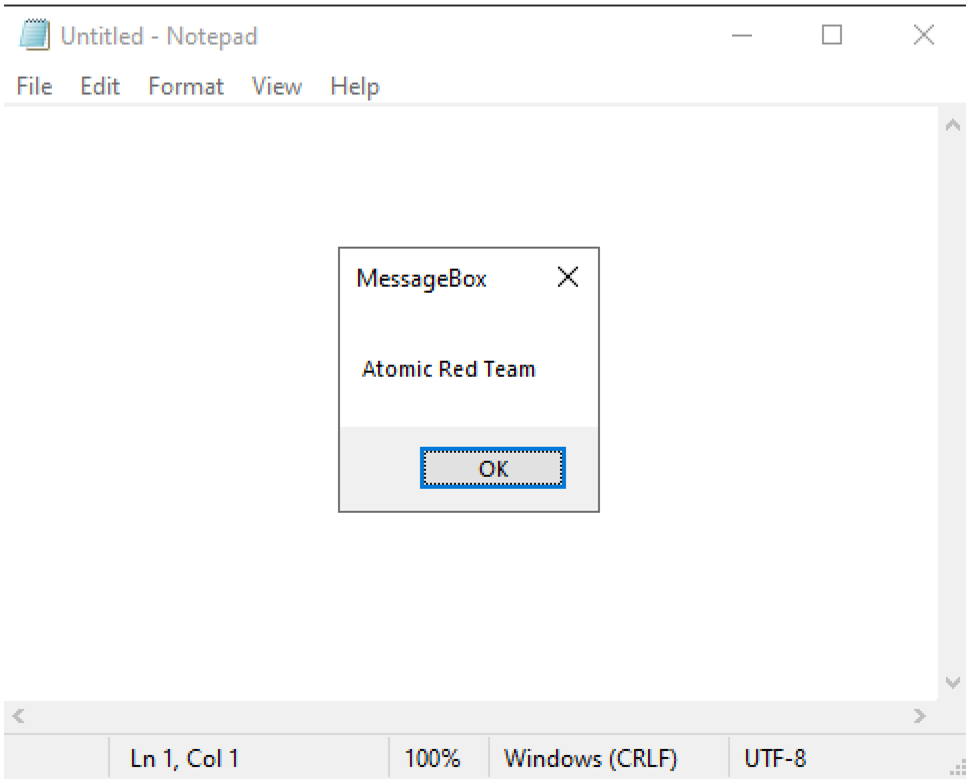
Thread Execution Hijacking
Also known as Suspend, Inject, Resume, the malware will first **suspend** the thread that runs a process, then use *VirtualAlloc* to regulate the process' memory before **injecting** a dll's path into that memory location. Then, a simple use of *LoadLibrary* will load the actual malicious code from that library and that code will instruct the thread to **resume**.

Process Hollowing
Similarly to the Thread Execution Hijacking, in Process Hollowing the malware suspends the thread, but instead of regulating the memory with *VirtualAlloc* to provide additional memory to the process, they use it to remove the process' original code. Then, with the *WriteProcessMemory* call, the malicious code will be added directly to that area of the memory, before signaling the thread to resume. The main difference with Thread Execution Hijacking is the fact that in Process Hollowing, the original process gets completely wiped and only its vessel remains to execute the malware instead of both the malware and the legitimate code running simultaneously.

Example
If we open a notepad and run the module T1055 from the Atomic Red Team (https://github.com/redcanaryco/atomic-red-team last day visited: 14/2/2023) package, we will see a messagebox appear at our notepad. This process hijacked a legitimate one to run code.



```
C:\Users\Flare VM>"c:\Users\Flare VM\Desktop\atomic-red-team-master\atomics\T1055\bin\x64\InjectView.exe"
Notepad.exe not running. Run Notepad first.

FLARE 09/02/2023  0:20:43.31
C:\Users\Flare VM>"c:\Users\Flare VM\Desktop\atomic-red-team-master\atomics\T1055\bin\x64\InjectView.exe"
```



Atomic Red Team

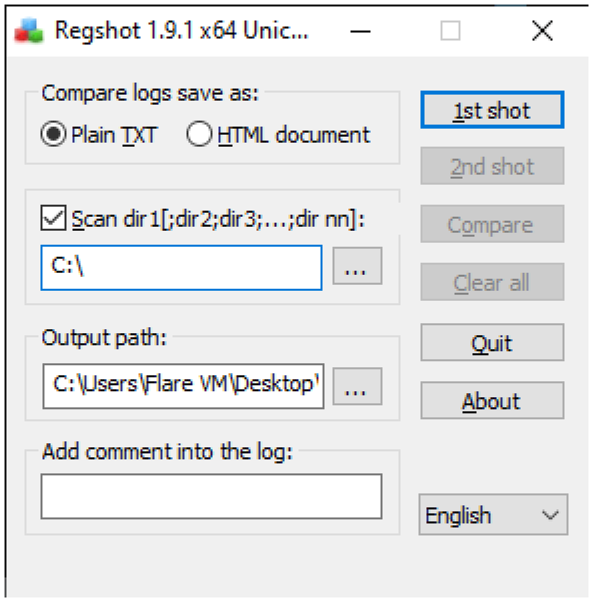
Figures 2.4.7 - Process Injection

Detonating Malware

Detonating malware is a potentially dangerous issue and analysts should take additional measures to prevent the malware from escaping, while still doing their work.
The first measure is by correctly setting up their Lab environment. Thankfully, almost all hypervisors have a screenshot feature where the user can save the state of a system and recall it. So, once the analysis is complete, going back to a previous state becomes rather easy.
The second measure is also for safety and revolves around networking. By setting up the gateway VM correctly, analysts can cut traffic to the machine used for analysis at will in case things get out of hand.
With the lab being already set up, we are ready for the most dangerous and most rewarding part of Malware Analysis, which is actually detonating malware.

enshRegistry and Filesystem Monitoring

In order to find what changes a malware did on our system, we need a reliable way to know the changes that took place. Our limited human memory is obviously not the best tool to handle such a task, but there is an old yet reliable tool that can do that for us. That tool is Regshot and its main function is to take a screenshot of the registry and the filesystem. After detonating the malware, we can run Regshot again to see what changes appeared in our system. The major drawback Regshot has is that it cannot possibly understand what changes were made due to the malware and what changes were scheduled Windows tasks. The only thing we can do is to reduce to the minimum the time between the two screenshots and do not perform any other tasks other than detonating the malware.



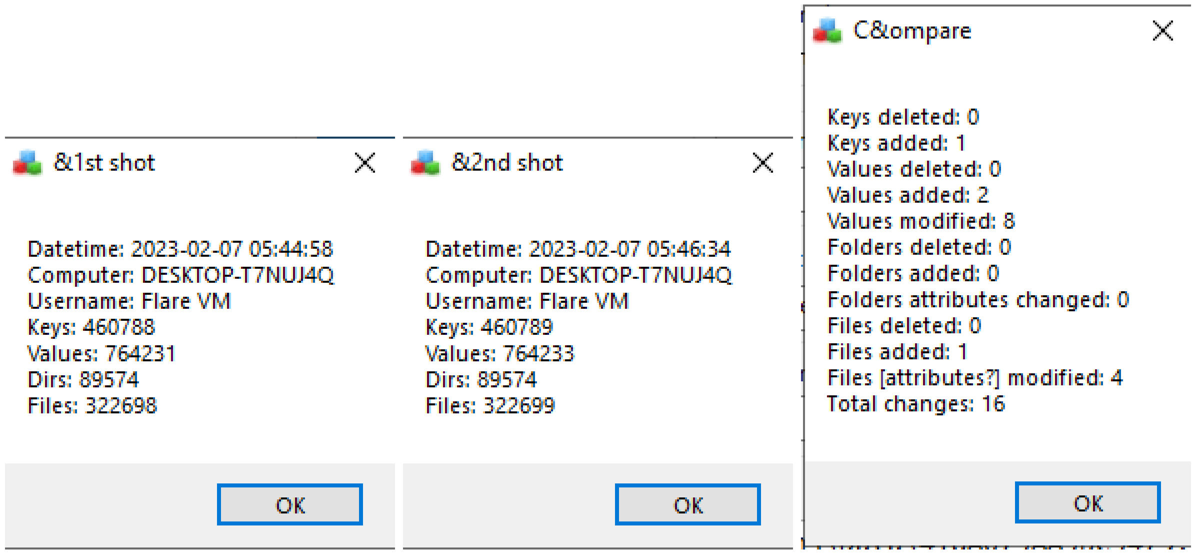
Compare logs save as: [✓ Plain TXT] HTML document

[✓] Scan dir1[;dir2;dir3;...;dir nn]: C:\

Output path: C:\Users\Flare VM\Desktop\

English

Figure 2.4.8 - Regshot User Interface



```
Datetime: 2023-02-07 05:44:58
Computer: DESKTOP-T7NUJ4Q
Username: Flare VM
Keys: 460788
Values: 764231
Dirs: 89574
Files: 322698
```

```
Datetime: 2023-02-07 05:46:34
Computer: DESKTOP-T7NUJ4Q
Username: Flare VM
Keys: 460789
Values: 764233
Dirs: 89574
Files: 322699
```

```
Keys deleted: 0
Keys added: 1
Values deleted: 0
Values added: 2
Values modified: 8
Folders deleted: 0
Folders added: 0
Folders attributes changed: 0
Files deleted: 0
Files added: 1
Files [attributes?] modified: 4
Total changes: 16
```

Figure 2.4.9 - Regshot before and after detonating the locky.exe ransomware

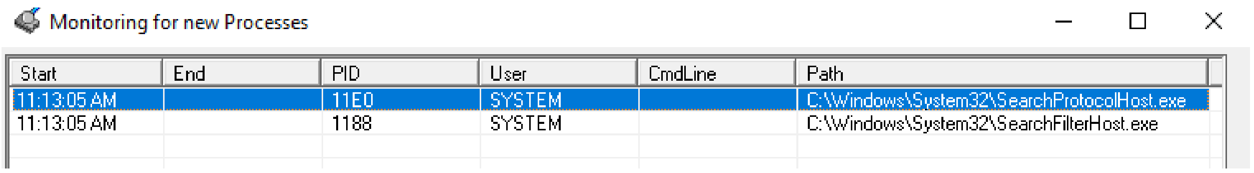
The locky.exe ransomware is a known and well documented threat, however it is known to evade detection. Still, Regshot found the modifications it did in less than two minutes. Time to restore Flare VM to a previous screenshot before proceeding.

Monitoring Processes

We have already seen how to identify processes. However, since we will be monitoring processes as they are created, we need tools that can ignore the old ones and display the recent or at least a tool that can conveniently filter them.

ProcWatch

Such a tool is ProcWatch, already implemented in Flare VM.



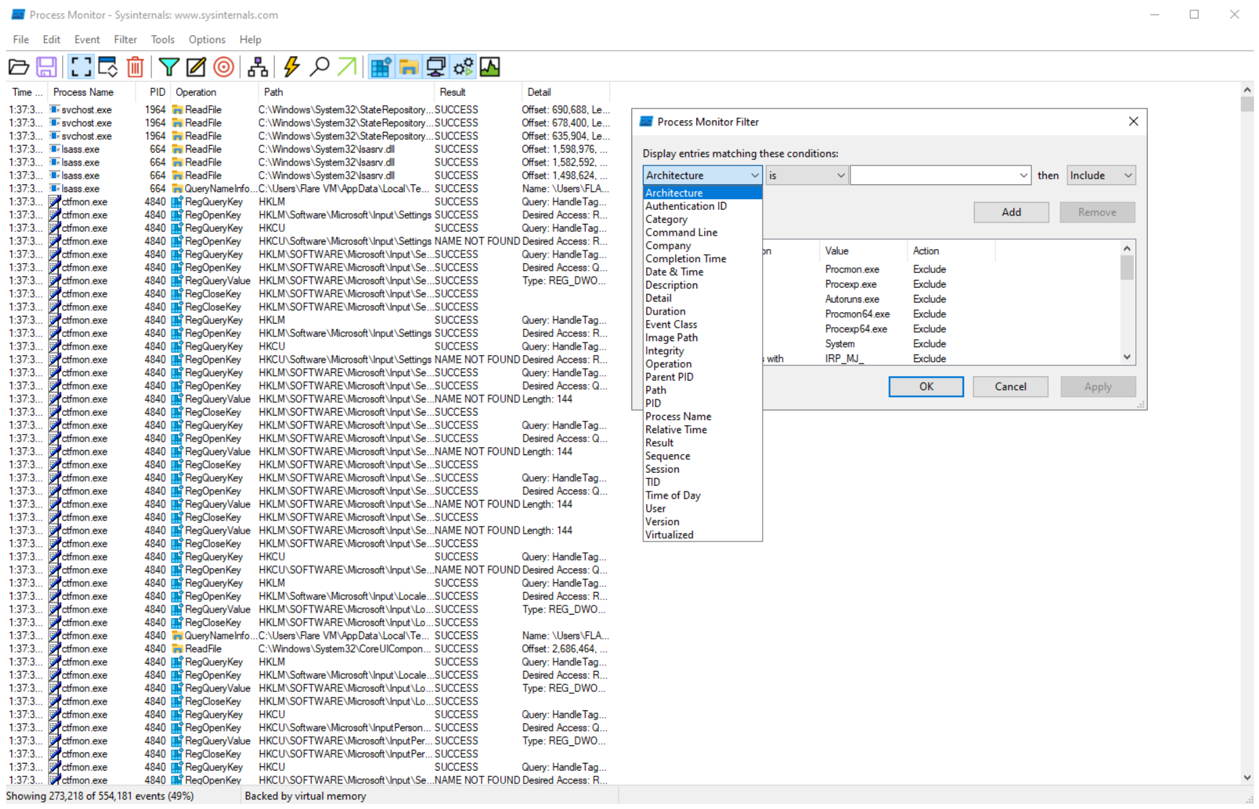
Start	End	PID	User	CmdLine	Path
11:13:05 AM		11E0	SYSTEM		C:\Windows\System32\SearchProtocolHost.exe
11:13:05 AM		1188	SYSTEM		C:\Windows\System32\SearchFilterHost.exe

Figure 2.4.10 - The ProcWatch tool

As we can see, the interface is really simple and the information displayed contains the timestamp of the process, the process ID, the user that initiated the process and the path of the file that created the process. It is worth noting that ProcWatch needs to be run as administrator in order to display processes not created by the active user. Since many malware usually try to get administrator privileges before executing the malicious part of the code, it is always better to run it as an administrator.
Getting the location of the file that created the process is the most important part of the entire process monitoring procedure, because once we have the file we can immediately start our reconnaissance utilizing static analysis techniques like hashing.

ProcMon

Another tool, far more sophisticated but less user-friendly is ProcMon. It needs some setup to get working but it can deliver immense information when detonating malware. It can monitor registry, filesystem, network, processes, threads and events. Also, it has a very well versed filtering system that allows us to cancel the noise that can exist by legitimate applications and services. For example, if we know that the malware is in the form of a dll, we can only display processes with the name regsvr32.exe or rundll32.exe and if we know that the malware is a vba script attached to a microsoft excel or a microsoft word file Excel.exe or Word.exe respectively.



```
Time ...  Process Name  PID   Operation        Path                                     Result          Detail
1:37:3... svchost.exe   1964  ReadFile         C:\Windows\System32\StateRepository...   SUCCESS         Offset: 690,688, Le...
1:37:3... svchost.exe   1964  ReadFile         C:\Windows\System32\StateRepository...   SUCCESS         Offset: 678,400, Le...
1:37:3... svchost.exe   1964  ReadFile         C:\Windows\System32\StateRepository...   SUCCESS         Offset: 635,904, Le...
1:37:3... lsass.exe      664  ReadFile         C:\Windows\System32\lsasrv.dll           SUCCESS         Offset: 1,598,976, ...
1:37:3... lsass.exe      664  ReadFile         C:\Windows\System32\lsasrv.dll           SUCCESS         Offset: 1,582,592, ...
1:37:3... lsass.exe      664  ReadFile         C:\Windows\System32\lsasrv.dll           SUCCESS         Offset: 1,498,624, ...
1:37:3... lsass.exe      664  QueryNameInfo... C:\Users\Flare VM\AppData\Local\Te...    SUCCESS         Name: \Users\FLA...
1:37:3... ctfmon.exe    4840  RegQueryKey      HKLM                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKLM\Software\Microsoft\Input\Settings   SUCCESS         Desired Access: R...
1:37:3... ctfmon.exe    4840  RegQueryKey      HKCU                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKCU\Software\Microsoft\Input\Settings   NAME NOT FOUND  Desired Access: R...
1:37:3... ctfmon.exe    4840  RegQueryKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Desired Access: Q...
1:37:3... ctfmon.exe    4840  RegQueryValue    HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Type: REG_DWO...
1:37:3... ctfmon.exe    4840  RegCloseKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS
1:37:3... ctfmon.exe    4840  RegCloseKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS
1:37:3... ctfmon.exe    4840  RegQueryKey      HKLM                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKLM\Software\Microsoft\Input\Settings   SUCCESS         Desired Access: R...
1:37:3... ctfmon.exe    4840  RegQueryKey      HKCU                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKCU\Software\Microsoft\Input\Settings   NAME NOT FOUND  Desired Access: R...
1:37:3... ctfmon.exe    4840  RegQueryKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Desired Access: Q...
1:37:3... ctfmon.exe    4840  RegQueryValue    HKLM\SOFTWARE\Microsoft\Input\Se...      NAME NOT FOUND  Length: 144
1:37:3... ctfmon.exe    4840  RegCloseKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS
1:37:3... ctfmon.exe    4840  RegQueryKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Desired Access: Q...
1:37:3... ctfmon.exe    4840  RegQueryValue    HKLM\SOFTWARE\Microsoft\Input\Se...      NAME NOT FOUND  Length: 144
1:37:3... ctfmon.exe    4840  RegCloseKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS
1:37:3... ctfmon.exe    4840  RegQueryKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS         Desired Access: Q...
1:37:3... ctfmon.exe    4840  RegQueryValue    HKLM\SOFTWARE\Microsoft\Input\Se...      NAME NOT FOUND  Length: 144
1:37:3... ctfmon.exe    4840  RegCloseKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS
1:37:3... ctfmon.exe    4840  RegQueryValue    HKLM\SOFTWARE\Microsoft\Input\Se...      NAME NOT FOUND  Length: 144
1:37:3... ctfmon.exe    4840  RegCloseKey      HKLM\SOFTWARE\Microsoft\Input\Se...      SUCCESS
1:37:3... ctfmon.exe    4840  RegQueryKey      HKCU                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKCU\SOFTWARE\Microsoft\Input\Se...      NAME NOT FOUND  Desired Access: Q...
1:37:3... ctfmon.exe    4840  RegQueryKey      HKLM                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKLM\Software\Microsoft\Input\Locale...  SUCCESS         Desired Access: R...
1:37:3... ctfmon.exe    4840  RegQueryValue    HKLM\SOFTWARE\Microsoft\Input\Lo...      SUCCESS         Type: REG_DWO...
1:37:3... ctfmon.exe    4840  RegCloseKey      HKLM\SOFTWARE\Microsoft\Input\Lo...      SUCCESS
1:37:3... ctfmon.exe    4840  QueryNameInfo... C:\Users\Flare VM\AppData\Local\Te...    SUCCESS         Name: \Users\FLA...
1:37:3... ctfmon.exe    4840  ReadFile         C:\Windows\System32\CoreUICompon...      SUCCESS         Offset: 2,686,464, ...
1:37:3... ctfmon.exe    4840  RegQueryKey      HKLM                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKLM\Software\Microsoft\Input\Locale...  SUCCESS         Desired Access: R...
1:37:3... ctfmon.exe    4840  RegQueryValue    HKLM\SOFTWARE\Microsoft\Input\Lo...      SUCCESS         Type: REG_DWO...
1:37:3... ctfmon.exe    4840  RegCloseKey      HKLM\SOFTWARE\Microsoft\Input\Lo...      SUCCESS
1:37:3... ctfmon.exe    4840  RegQueryKey      HKCU                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKCU\Software\Microsoft\InputPerson...   SUCCESS         Desired Access: Q...
1:37:3... ctfmon.exe    4840  RegQueryValue    HKCU\SOFTWARE\Microsoft\InputPer...      SUCCESS         Type: REG_DWO...
1:37:3... ctfmon.exe    4840  RegCloseKey      HKCU\SOFTWARE\Microsoft\InputPer...      SUCCESS
1:37:3... ctfmon.exe    4840  RegQueryKey      HKCU                                     SUCCESS         Query: HandleTag...
1:37:3... ctfmon.exe    4840  RegOpenKey       HKCU\SOFTWARE\Microsoft\Input\Se...      NAME NOT FOUND  Desired Access: R...
Showing 273,218 of 554,181 events (49%)    Backed by virtual memory
```

Process Monitor Filter

Display entries matching these conditions:

Architecture | is | | then | Include

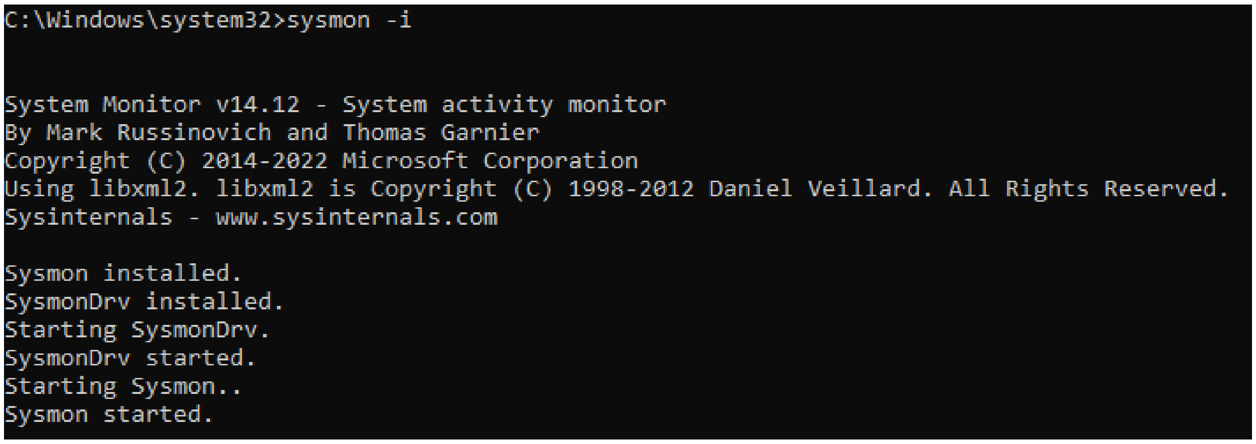
```
...on     Value           Action
          Procmon.exe     Exclude
          Procexp.exe     Exclude
          Autoruns.exe    Exclude
          Procmon64.exe   Exclude
          Procexp64.exe   Exclude
          System          Exclude
...with   IRP_MJ_         Exclude
```

Figure 2.4.11 - Procmon and its sophisticated filter system.

Monitoring Process Injection

On Process Injection, as we studied them, it is obvious that the tools we have seen so far are helpful but not reliable. There is however a tool we can use named Sysmon (System Monitor). To install it, we should simply open the command prompt as administrators and type

sysmon -i



```
C:\Windows\system32>sysmon -i


System Monitor v14.12 - System activity monitor
By Mark Russinovich and Thomas Garnier
Copyright (C) 2014-2022 Microsoft Corporation
Using libxml2. libxml2 is Copyright (C) 1998-2012 Daniel Veillard. All Rights Reserved.
Sysinternals - www.sysinternals.com

Sysmon installed.
SysmonDrv installed.
Starting SysmonDrv.
SysmonDrv started.
Starting Sysmon..
Sysmon started.
```

Figure 2.4.12 - Sysmon installation and initiation

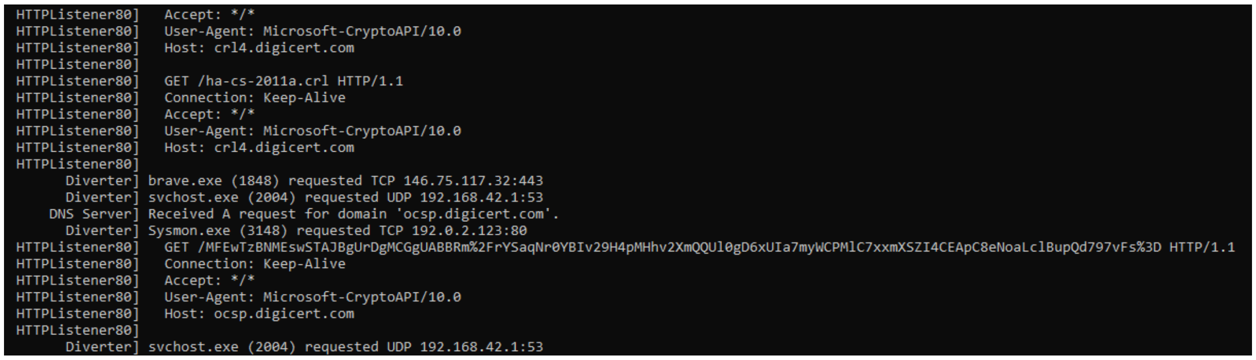
Now, to monitor whether or not there was a process injection event, we need to open the Event Viewer and go to

Applications and Service Logs\Microsoft\Windows\Sysmon\Operational

and filter for an event with ID 8. It is the log generated when *CreateRemoteThread* is called. According to Microsoft's sysmon documentation (https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon last day visited: 14/2/2023), "The CreateRemoteThread event detects when a process creates a thread in another process."

Monitoring Network Traffic

We already saw how to monitor traffic using wireshark at the gateway system. However, there are use cases where we want to remain inside our FlareVM. One such case may be that the malware is suspected of being able to move between nodes of a network. In order to restrict such behavior, we can contain the traffic inside Flare VM and use tools to monitor it. A tool, already included in Flare VM, that allows us to do so is FakeNet-NG. While not the most user friendly application, it is highly efficient as it intercepts, records and displays traffic, by creating a virtual adapter and forcing all traffic through it.



```
HTTPListener80]   Accept: */*
HTTPListener80]   User-Agent: Microsoft-CryptoAPI/10.0
HTTPListener80]   Host: crl4.digicert.com
HTTPListener80]
HTTPListener80]   GET /ha-cs-2011a.crl HTTP/1.1
HTTPListener80]   Connection: Keep-Alive
HTTPListener80]   Accept: */*
HTTPListener80]   User-Agent: Microsoft-CryptoAPI/10.0
HTTPListener80]   Host: crl4.digicert.com
HTTPListener80]
      Diverter] brave.exe (1848) requested TCP 146.75.117.32:443
      Diverter] svchost.exe (2004) requested UDP 192.168.42.1:53
    DNS Server] Received A request for domain 'ocsp.digicert.com'.
      Diverter] Sysmon.exe (3148) requested TCP 192.0.2.123:80
HTTPListener80]   GET /MFEwTzBNMEswSTAJBgUrDgMCGgUABBRm%2FrYSaqNr0YBIv29H4pMHhv2XmQQUl0gD6xUIa7myWCPMlC7xxmXSZI4CEApC8eNoaLclBupQd797vFs%3D HTTP/1.1
HTTPListener80]   Connection: Keep-Alive
HTTPListener80]   Accept: */*
HTTPListener80]   User-Agent: Microsoft-CryptoAPI/10.0
HTTPListener80]   Host: ocsp.digicert.com
HTTPListener80]
      Diverter] svchost.exe (2004) requested UDP 192.168.42.1:53
```

Figure 2.4.13 - FakeNet-NG Command Line Interface

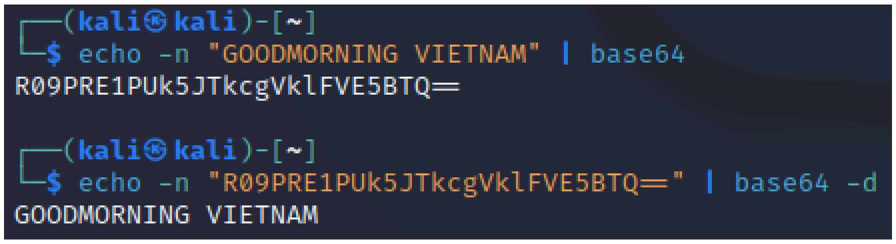
De-Obfuscating Malware

Before starting this chapter, we must realize that there is no standard method for de-obfuscating code. Here, we will view techniques used universally for obfuscating code and the way to counter them, followed by some techniques used for obfuscating powershell code. Choosing powershell because it is one of the most potent ways to create malware and can be executed in all machines running a Windows or a Mac Operating System. Powershell can also run out of the box on various Linux distributions including the most popular ones (Ubuntu, CentOS, Fedora, Debian, Red Hat)

Encoding

Encoding is the most obvious first step for making the code harder to analyze, as it gets transformed to a non-human readable language. Fortunately, encoding is not by any means encrypted. And thus, the procedure is reversible. There are various ways to encode, with the most common ones being:

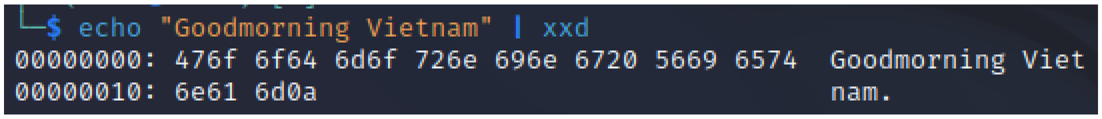
- Base64 (and variations): Base64 is a scheme that can encode text to binary and vice versa. An indication that something is encoded under Base64 is the fact that the encoded string has always a number of characters divisible by 4. If needed, the encoder itself adds the character "=" as many times as needed. Malicious actors are known to get out of their way to ensure that there are no such symbols in order to better avoid detection.
  Variations of Base64 are Base32, Base58, Base62 and Base85, depending on the characters the person doing the obfuscation desires.



```
┌──(kali㉿kali)-[~]
└─$ echo -n "GOODMORNING VIETNAM" | base64
R09PRE1PUk5JTkcgVklFVE5BTQ==

┌──(kali㉿kali)-[~]
└─$ echo -n "R09PRE1PUk5JTkcgVklFVE5BTQ==" | base64 -d
GOODMORNING VIETNAM
```

Figure 2.5.1 - Encoding and decoding a string under Base64

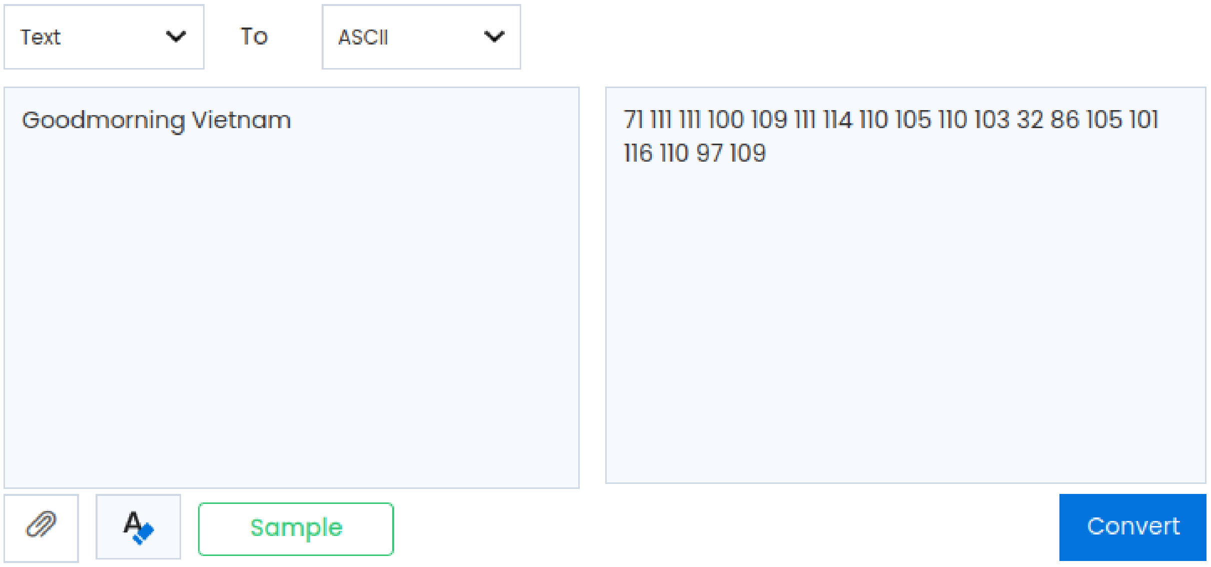
- Hexadecimal encoding: Hex encoding is easy to recognise, considering that there are no letters beyond the letter 'f' in hexadecimal notation, but remains hard to understand. We saw hexadecimal being used when viewing the contents of a file earlier, in Static Analysis.



```
└─$ echo "Goodmorning Vietnam" | xxd
00000000: 476f 6f64 6d6f 726e 696e 6720 5669 6574  Goodmorning Viet
00000010: 6e61 6d0a                                nam.
```

Figure 2.5.2 - Encoding a string under hex

- ASCII encoding: The ASCII table is widely used in many software applications and malware obfuscation is not an exception. There are many online tools capable of converting text to ascii and vice versa.



Goodmorning Vietnam

71 111 111 100 109 111 114 110 105 110 103 32 86 105 101 116 110 97 109

Figure 2.5.3 - Converting text to ascii

<u>Unconventional Variable and Function Naming</u>

Unsurprisingly, a very common method used to make code harder to read is variable and function naming that are unconventional. Almost all text editors have the ability to mass change a name to something that does not make much sense as a way to confuse anyone willing to look at it.

For example:

IIIlIlI and IIllll

seem extremely similar and yet, through combining the capital letter i and the lower case L, the code becomes much less readable. Another similar method is having long strings of random letters, each representing a different function or variable.

However, no matter how annoying this method of obfuscation is, it is still reversible by the same mechanisms that the malicious actor used. A cautious and persistent analyst will be able to de-obfuscate it.

<u>Clutter</u>

A common addition to the unconventional variable and function naming is the addition of junk lines. Variables and functions that do nothing other than confuse the analyst of their purpose. These often include variables that are set and used and functions that are called but with neither of them affecting the outcome of the overall program whatsoever. In some cases, clutter has been observed to be disproportionately more than the actual payload, sometimes reaching thousands of lines of useless code. And of course, most of the time, there are useless variables in useless functions.
Useless functions may include a long sequence of commands that always return true. Inserting that function in an if statement will have literally zero impact on the functionality of the malware.

<u>String Manipulation</u>

String manipulation almost always results in a different hash calculation, resulting in the malware being invisible to that method of analysis. Additionally, shell code like Powershell can have strings as execution arguments resulting in this method being both potent and easy to perform.
There are two major ways for a hacker to manipulate strings in order to obfuscate code.

- Replacement: By injecting specific character sequences and having the malware remove them before execution, the author of the malware manages to only run the “useful” part of the code.

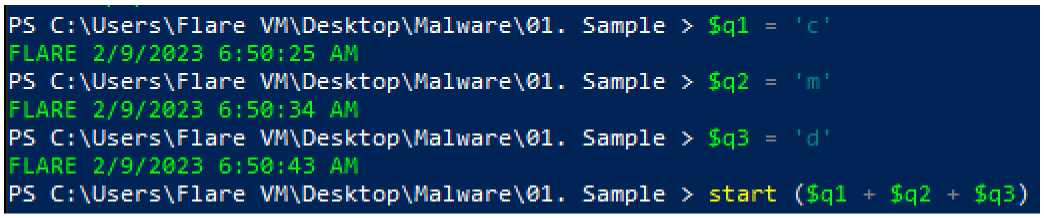
  For example, if they want to open a new Command Line they can run it like this:

  ```
  $qwe = “cHGVKJHVGBUKJHmHGVKJHVGBUKJHdHGVKJHVGBUKJH”
  start (($qwe -Replace “HGVKJHVGBUKJH” “”))
  ```

  What remains as an argument is: “cmd”

- Concatenation: By breaking the string to many parts and linking them together before execution, a similar result is achieved

  For the same example:



```
PS C:\Users\Flare VM\Desktop\Malware\01. Sample > $q1 = 'c'
FLARE 2/9/2023 6:50:25 AM
PS C:\Users\Flare VM\Desktop\Malware\01. Sample > $q2 = 'm'
FLARE 2/9/2023 6:50:34 AM
PS C:\Users\Flare VM\Desktop\Malware\01. Sample > $q3 = 'd'
FLARE 2/9/2023 6:50:43 AM
PS C:\Users\Flare VM\Desktop\Malware\01. Sample > start ($q1 + $q2 + $q3)
```

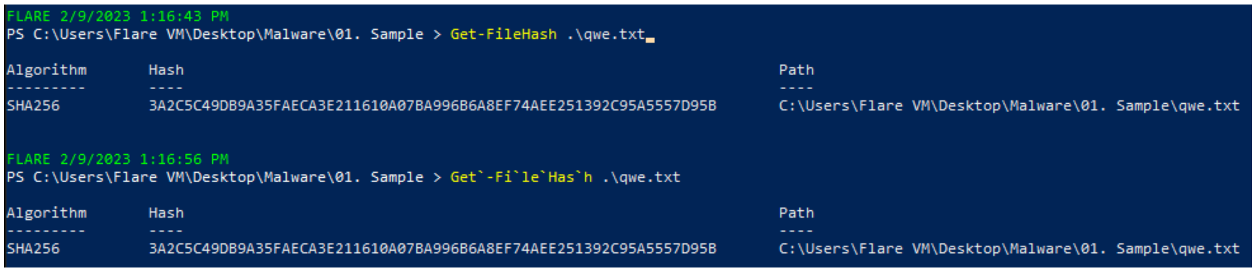
Figure 2.5.4 - Starting cmd by concatenating the arguments of a function

Will result in the argument (cmd.exe) and thus opening a new Command Line window.

- Array Element Reordering: A common addition to concatenation is reordering with the use of arrays. By splitting the name of command and placing the resulting strings in an array in random positions, the malware authors can hide the actual commands in plain sight. More arrays and more elements further increase the complexity.

<u>Backticks (Powershell Specific)</u>

Commands in Powershell can usually be separated with the backtick character ‘ ` ’ without really disrupting the execution of the code.



```
FLARE 2/9/2023 1:16:43 PM
PS C:\Users\Flare VM\Desktop\Malware\01. Sample > Get-FileHash .\qwe.txt

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          3A2C5C49DB9A35FAECA3E211610A07BA996B6A8EF74AEE251392C95A5557D95B       C:\Users\Flare VM\Desktop\Malware\01. Sample\qwe.txt


FLARE 2/9/2023 1:16:56 PM
PS C:\Users\Flare VM\Desktop\Malware\01. Sample > Get`-Fi`le`Has`h .\qwe.txt

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          3A2C5C49DB9A35FAECA3E211610A07BA996B6A8EF74AEE251392C95A5557D95B       C:\Users\Flare VM\Desktop\Malware\01. Sample\qwe.txt
```

Figure 2.5.4 - Separating Powershell Code with backticks

<u>Whitespace (Powershell Specific)</u>

In powershell, and any application created under the .NET framework, whitespace does not really matter, however it does provide a potent obfuscation method when combined with other methods. For example, along with string concatenation:

(‘New-Object’) is the same as (‘Ne’   +’w-O’ +   ‘bj’ +   ‘ec’   +   ‘t’)

<u>Combinations</u>

As we already described in each of the previous categories, it is relatively easy to deobfuscate code when only one method has been used. It gets significantly harder to borderline impossible when these methods get combined though, since we, as analysts, need to find out the exact steps that the malware's author used and reverse them in the correct order. This mostly comes through trial and error, though an experienced analyst could potentially understand the sequence faster.

Malware Analysis Workflow

We talked a lot in the entire malware analysis section about various tools that are regularly used by analysts in order to perform their duties. It is now time to put everything together in a workflow that will consistently yield results. This workflow can be divided into four main parts.

Information Gathering

Before actually running the malware, we need to check it statically. If the malware is known, we could run it for research or for practice, but in most cases the cybersecurity community will have already done so and will have posted any findings online.

1. We identify the type of file using the file command.
2. We calculate the hash value and compare it on the various databases available online.
3. We view the strings included in the file, they may contain API calls or even information like the malware name or the author or a list of tools that their mere presence would disrupt the malware's detonation.
4. We analyze the Portable Executable file system with CFF Explorer for anything useful.
5. If nothing makes sense, we might be facing a packer. We should check the Entropy with the appropriate tool and if it is high enough, we could try to decompress it with various algorithms and check again. If not, we are either facing an obfuscated thread or a novelty.

In case the malware is obfuscated, we have to try reverting it to a human-readable state in order to further analyze it statically. Else, we will move on to Dynamic Analysis.

Detonating the Malware

1. First, we start FakeNet-NG.
2. Secondly, we run ProcMon. We immediately pause and clear it to remove clutter. We pause it because ProcMon captures almost anything that happens, even useful and legitimate processes. And on our next step, RegShot is going to create an immense amount of records. We do not want them.
3. We run Regshot and take the first screenshot. We should screenshot the entirety of the System drive (assuming C: for the majority of cases). After it is done, we can resume ProcMon.
4. Now is the time to run the malware and view the FakeNet-NG generated output. It is important to pay attention to the network connectivity, in order to capture as it

happens any attempts to contact a Command and Control server or download any additional tools, libraries or files.

5. We pause ProcMon and use Regshot to take the second screenshot.

There are a few points we must keep in mind.

- Some malware have a long wait time remaining dormant specifically to avoid detection. While some will run instantly and only need seconds, some others can lie and wait for 30 minutes or even more, according to their sophistication level. As a starting point, we can run malware for 3-5 minutes and, if needed, we can reset our Virtual Machine to a previous state and let it run for more or less according to the results.
- If a piece of malware tries to download additional files or tools, after capturing the addresses it tries to visit, we should reset the entire process (including reverting our Flare VM to an older state) and run the entire process again WITHOUT starting FakeNet-NG. We do that to simulate a real world scenario where all the malware files are present. In these cases, we should get all the traffic through the gateway with the use of Wireshark.
- Running a simple process like notepad or calculator as administrator will give the malware the opportunity to hijack processes with elevated privileges, so we can study in real time its capabilities.

It is clear by now that we want to provide the malware the appropriate breeding ground, in order for it to cause the maximum damage. While counter intuitive, malware analysts should always assume the worst case scenario and exhaust the malware's capabilities.

Reviewing the Results

1. We open the Regshot comparison output.
2. We apply a few filters on ProcMon in order to focus on whatever happened. The main operations we are interested in are:
    - TCP and UDP Connections (TCP Connect and UDP Connect)
    - Registry values set (RegSetValue)
    - Processes Created (Process Create)
    - Files Created (WriteFile)
    - Files Deleted (SetDispositionInformationFile)

   Of course, we are also interested in any process named (ProcessName) after the malware.
3. We open the Event Viewer and study the SysMon logs (assuming we have already installed and enabled SysMon). We look for any events with ID equal to 8, as these are the processes that indicate Process Injection

Neutralizing the Threats and Disclosing the Findings

The analyst probably has some kind of obligation with the organization they work on. They need to neutralize the threat, whether that be blocking adversarial IPs at the firewall or excluding operations from the machines. Completely setting all systems from scratch is the safest approach, however it is not always a viable one. In these cases, the analyst's job revolves around securing the infected systems as to prevent the malware from spreading before setting new policies for all the computers in the network or the organization.

In all cases, the cybersecurity academic and professional community must be informed of the analyst's findings as soon as possible. Disclosing the findings is more of a legal procedure and involves working with the appropriate organizations involved, but it is important for the malware to be documented for the protection of everyone using any sort of device.

Reverse Engineering

What is Reverse Engineering in Malware Analysis

Reverse Engineering in general is the process of analyzing the results that an existing product produces with the goal to design and create another product that can replicate these results.
When it comes to Malware Analysis, it boils down to figuring out the source code and the way it interacts with the system on a more primitive level, like the way it handles memory and the way it interacts with the processor. The ultimate goal is to isolate the malware to a single binary file that is stripped of other legitimate operations.
In this chapter we will first establish some basic terminology before delving into the tools used for Reverse Engineering.

The Basics

Though the Assembly language, the Stack and the Heap as well as the Memory Management are way beyond the scope of this thesis, some basics need to be explained in order to better understand the tools and techniques of this section.

CPU Registers

The registers are small parts of data (typically consisting of 64 bits in modern systems) that are part of the processor and either hold data or instructions. Due to their limited size, registers cannot possibly contain files, however are more than capable of storing memory addresses, pointers and instructions (functions).

The important registers are:

- RSP - Stack Pointer, a pointer to the top of the stack where the next instruction to be executed resides (more on the stack later)
- RIP - Instruction Pointer, pointing to the memory address of the next instruction to be executed. RIP can point to the top of the stack (and thus having the same value as RSP), however it can also have a completely different value if points to a function call
- RAX - Accumulator, stores the results of API calls

Other typical Registers in an x64 system include:

RBX - Base, is a pointer to data that is going to be used

RCX - Counter, used for shifting instructions and for looping
RDX - Data, whether that be arithmetic or input/output
RBP - Base Pointer, a pointer to the base of the stack (more on the stack later)
RSI - Source Index, a pointer to source in strings
RDI - Destination Index, a pointer to destination in strings

Since we will be analyzing malware and most malware is designed to run on both 64 bit and 32 bit processors, it is important to remember that registers starting with the letter R indicate 64 bit architecture and with the letter E, 32 bit architecture.

Microsoft's documentation is a great resource on both the x64 (64 bit) and the x86 (32 bit) architectures and can be found here:
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture
(last day visited - 14/2/2023)

The Stack

Stack, commonly referred to as LIFO (Last In First Out), is one of the most primitive Data Structures. When it comes to memory management, The Stack is a continuous block of memory. Its purpose is to store local variables and return the addresses of various functions. It is managed through the push, pop, call and ret operations.

The Heap

Heap is a binary tree based data structure. When it comes to memory management, The Heap's purpose is to store global store variables created dynamically during the execution of a program.

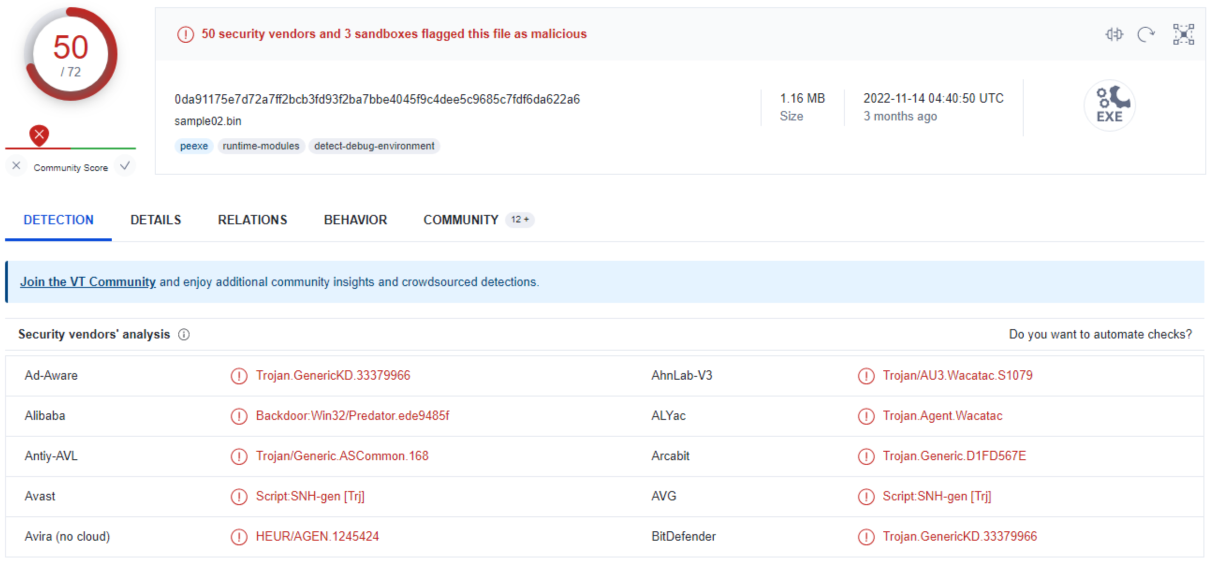
Assembly Instructions

There are various sets of instructions, each serving different functionality. The most prominent ones are:

- Logic: These instructions perform operations the same way a logic gate would. This set of instructions includes and, or, xor, not, add, etc.
- Control flow: These instructions can change the flow of execution by moving to a different section. They are also used for comparisons. This set includes push, call, jne, jnz, jz, jmp, etc
- Data Transfer: The instruction mov is used to transfer data stored in registers.
- No Operation: nop is probably the most notorious operation for a good reason, as it has been repeatedly used to exploit memory and lead to unexpected operations. As an instruction, it simply tells the processor to do nothing and move to the next instruction.

Accessing the Source Code with Ghidra

Ghidra is a tool developed by United State's National Security Agency and has become one of the most commonly used tools both for Malware Analysis and for Reverse Engineering. It was leaked by WikiLeaks in 2017 and was declassified and released as open source in 2019. Ghidra can decompile, debug and provide the source code of an executable. As with any open source tool, it is free and can be downloaded from the official website https://www.ghidra-sre.org/ (last day visited - 14/2/2023).
In order to better display Ghidra's capabilities, we will use a trojan horse named Lokibot.



50 security vendors and 3 sandboxes flagged this file as malicious

0da91175e7d72a7ff2bcb3fd93f2ba7bbe4045f9c4dee5c9685c7fdf6da622a6
sample02.bin
peexe runtime-modules detect-debug-environment

1.16 MB Size
2022-11-14 04:40:50 UTC 3 months ago

Join the VT Community and enjoy additional community insights and crowdsourced detections.

Security vendors' analysis — Do you want to automate checks?

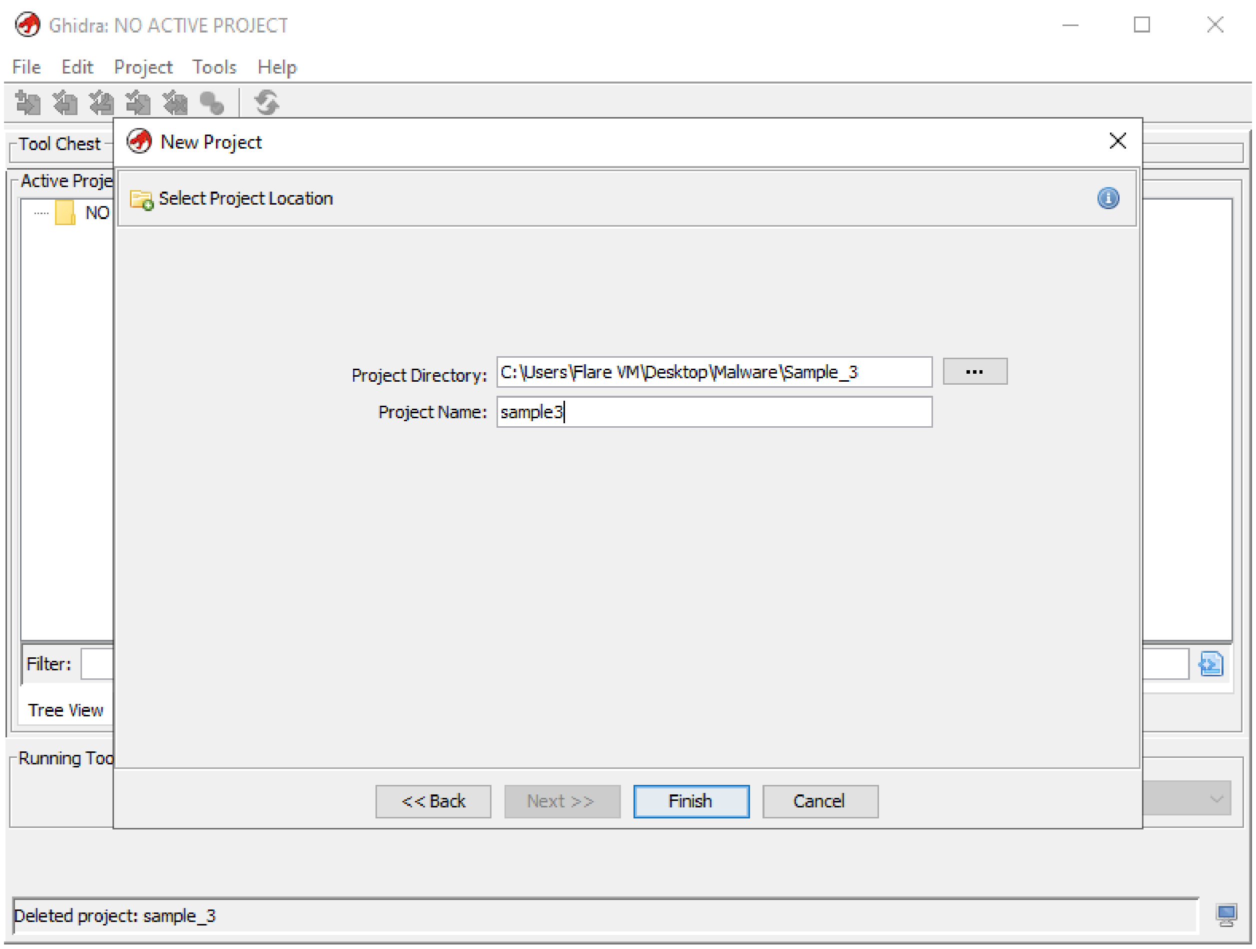
Ad-Aware	Trojan.GenericKD.33379966	AhnLab-V3	Trojan/AU3.Wacatac.S1079
Alibaba	Backdoor:Win32/Predator.ede9485f	ALYac	Trojan.Agent.Wacatac
Antiy-AVL	Trojan/Generic.ASCommon.168	Arcabit	Trojan.Generic.D1FD567E
Avast	Script:SNH-gen [Trj]	AVG	Script:SNH-gen [Trj]
Avira (no cloud)	HEUR/AGEN.1245424	BitDefender	Trojan.GenericKD.33379966

Figure 2.6.1 - The Lokibot trojan horse

Obviously, Lokibot is malicious and so all the precaution measures we talked about in the Malware Analysis section will be met.

Initiating the Process

First of all, we need to create a new Ghidra project. Will select Non-Shared for this occasion (A shared project enables the collaboration of many analysts in a similar way git enables it for developers).



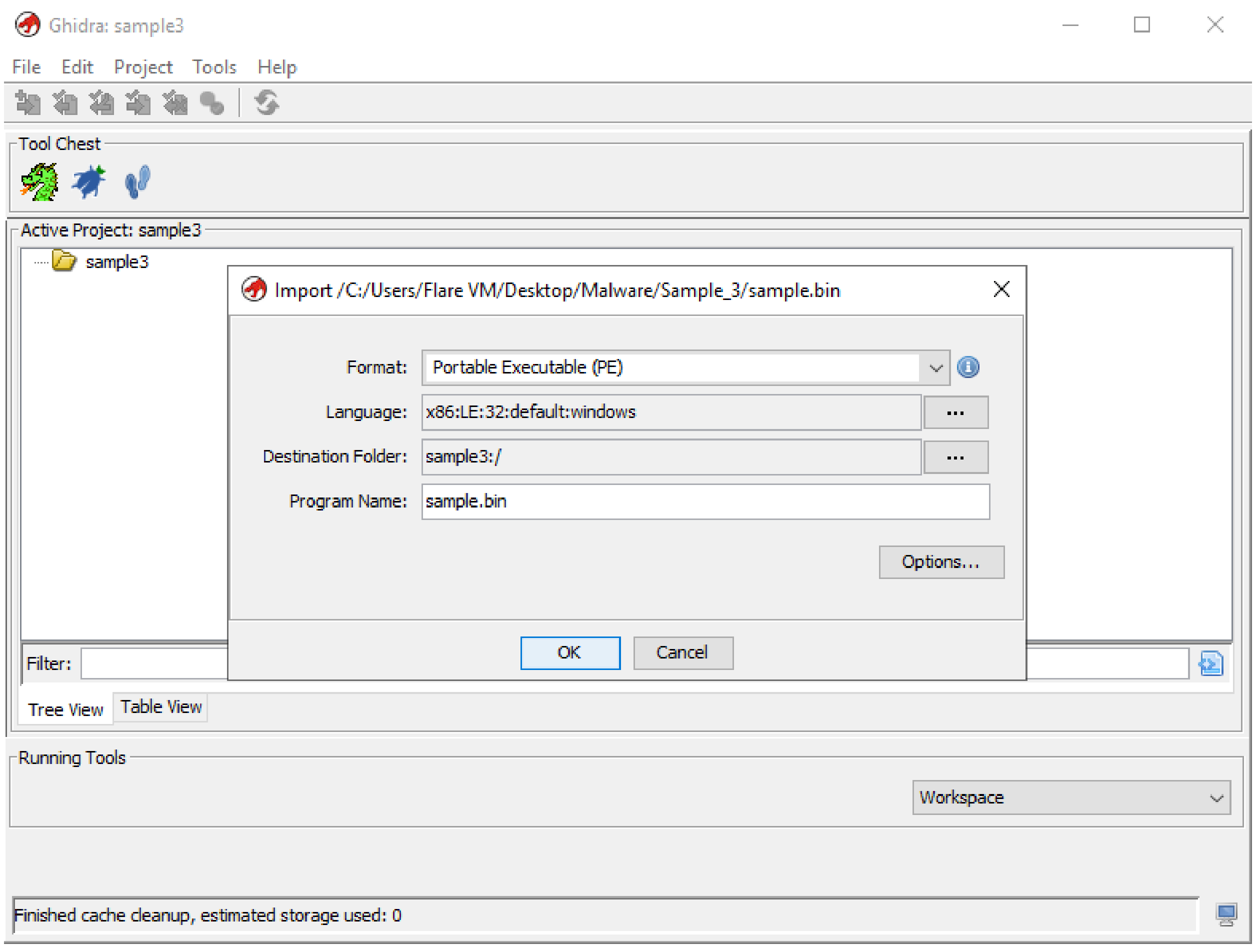
Project Directory: C:\Users\Flare VM\Desktop\Malware\Sample_3

Project Name: sample3

Deleted project: sample_3

Figure 2.6.2 - Creating new Ghidra Project

Simply dragging and dropping the file in the Ghidra environment will automatically result in us getting some information. On this occasion we immediately get notified that the file is of the Portable Executable format and that it is a 32 bit application for Windows systems.



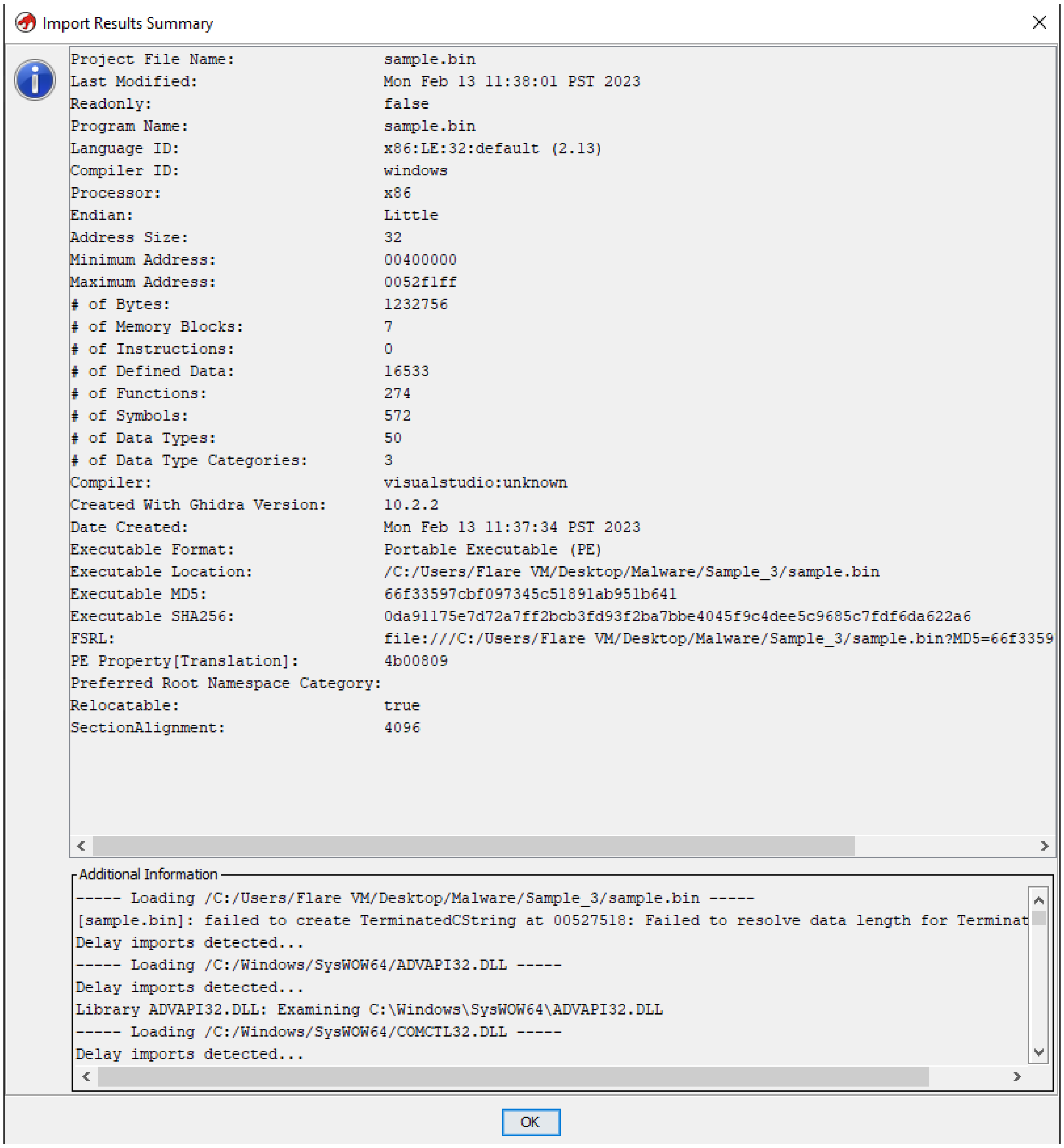
Import /C:/Users/Flare VM/Desktop/Malware/Sample_3/sample.bin

Format: Portable Executable (PE)
Language: x86:LE:32:default:windows
Destination Folder: sample3:/
Program Name: sample.bin

Finished cache cleanup, estimated storage used: 0

Figure 2.6.3 - Immediate Ghidra information

Once we click ok, Ghidra will initiate the disassembly of the executable. This process may take a while, according to the resources an analyst has allocated to the Virtual Machine. Once it is done, we get the summary of the procedure.
This summary goes much more in depth than what we got earlier and provides not only the target system and the architecture but also the size and the memory the executable will use if run.



```
Project File Name:                  sample.bin
Last Modified:                      Mon Feb 13 11:38:01 PST 2023
Readonly:                           false
Program Name:                       sample.bin
Language ID:                        x86:LE:32:default (2.13)
Compiler ID:                        windows
Processor:                          x86
Endian:                             Little
Address Size:                       32
Minimum Address:                    00400000
Maximum Address:                    0052f1ff
# of Bytes:                         1232756
# of Memory Blocks:                 7
# of Instructions:                  0
# of Defined Data:                  16533
# of Functions:                     274
# of Symbols:                       572
# of Data Types:                    50
# of Data Type Categories:          3
Compiler:                           visualstudio:unknown
Created With Ghidra Version:        10.2.2
Date Created:                       Mon Feb 13 11:37:34 PST 2023
Executable Format:                  Portable Executable (PE)
Executable Location:                /C:/Users/Flare VM/Desktop/Malware/Sample_3/sample.bin
Executable MD5:                     66f33597cbf097345c51891ab951b641
Executable SHA256:                  0da91175e7d72a7ff2bcb3fd93f2ba7bbe4045f9c4dee5c9685c7fdf6da622a6
FSRL:                               file:///C:/Users/Flare VM/Desktop/Malware/Sample_3/sample.bin?MD5=66f3359
PE Property[Translation]:           4b00809
Preferred Root Namespace Category:
Relocatable:                        true
SectionAlignment:                   4096
```

Additional Information

```
----- Loading /C:/Users/Flare VM/Desktop/Malware/Sample_3/sample.bin -----
[sample.bin]: failed to create TerminatedCString at 00527518: Failed to resolve data length for Terminat
Delay imports detected...
----- Loading /C:/Windows/SysWOW64/ADVAPI32.DLL -----
Delay imports detected...
Library ADVAPI32.DLL: Examining C:\Windows\SysWOW64\ADVAPI32.DLL
----- Loading /C:/Windows/SysWOW64/COMCTL32.DLL -----
Delay imports detected...
```

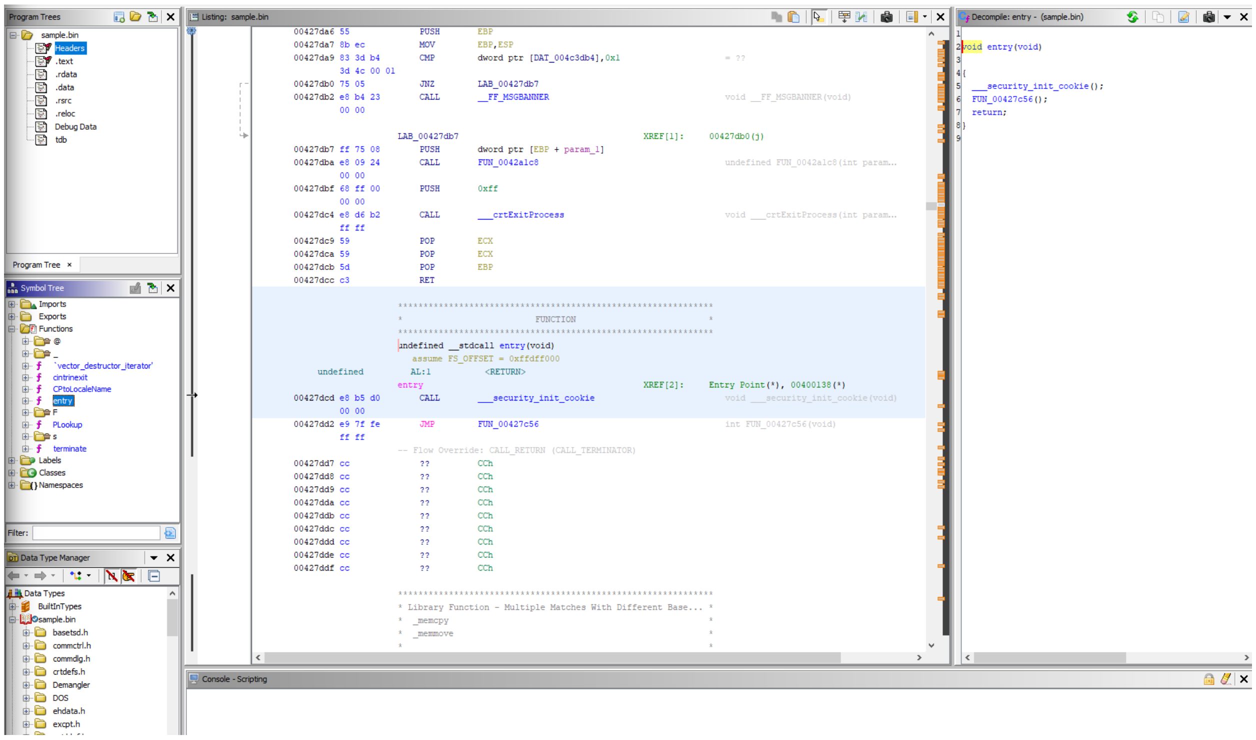
Figure 2.6.4 - Ghidra's Import Results Summary

Pressing ok will return us to the main window, with many options now activated. The first option in the Tool Chest panel is the Code Browser. Dragging and dropping the file on this button will create a new window and will prompt us to analyze the file (If it is the first time we opened the Code Browser for this specific file). Note that simply pressing the button does not work, we need to drag and drop the file. Pressing the yes button will open a new window with various options. We don't need to change anything here since

all functions not in prototype state will already be included. Pressing analyze will initiate the decompiling process. This will take a few minutes. Once done, we can start dissecting the file.
On the top left corner we can view all the sections of the Portable Executable as we studied them in the corresponding Malware Analysis chapter. On the middle left panel, we have the Symbol Tree that contains all the names that are found in the malware we are analyzing, including variables, functions and API calls. On the bottom left corner is the Data Type Manager, a convenient area where we can view all the known built in functions.
In the middle is the Listing, the main panel that provides us the code in Assembly.
Finally, on the right side is the Decompile window, the area where we can view the code in C. Note that this area is not always accurate, and needs to be treated with caution in order to understand what the malware does.

Starting Point

In order to begin actually analyzing this malware, we need to start from the beginning. In every programming language, there is an entry function. In C and C++ it is “int main()”, in Rust it is “fn main()”, in Java it is “public static void main(String[] args)”. When we go deeper though, in Assembly language, we only have the entry point. And Ghidra provides us with this entry point by simply finding the “entry” record under Functions in the Symbol Tree.

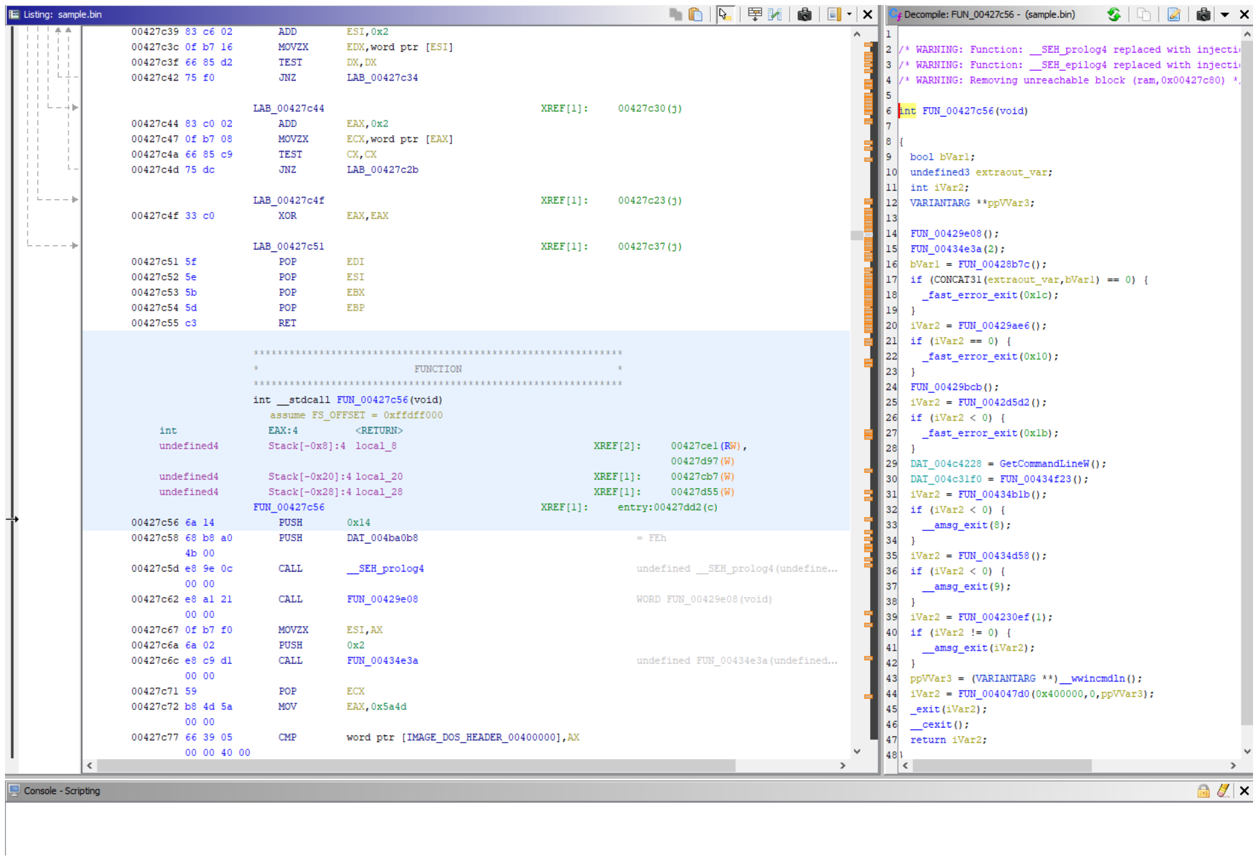


```
        00427da6 55              PUSH       EBP
        00427da7 8b ec           MOV        EBP,ESP
        00427da9 83 3d b4        CMP        dword ptr [DAT_004c3db4],0x1                    = ??
                 3d 4c 00 01
        00427db0 75 05           JNZ        LAB_00427db7
        00427db2 e8 b4 23        CALL       __FF_MSGBANNER                                  void __FF_MSGBANNER(void)
                 00 00

                             LAB_00427db7                             XREF[1]:     00427db0(j)
        00427db7 ff 75 08        PUSH       dword ptr [EBP + param_1]
        00427dba e8 09 24        CALL       FUN_0042a1c8                                    undefined FUN_0042a1c8(int param...
                 00 00
        00427dbf 68 ff 00        PUSH       0xff
                 00 00
        00427dc4 e8 d6 b2        CALL       ___crtExitProcess                               void ___crtExitProcess(int param...
                 ff ff
        00427dc9 59              POP        ECX
        00427dca 59              POP        ECX
        00427dcb 5d              POP        EBP
        00427dcc c3              RET

                             **************************************************************
                             *                          FUNCTION                          *
                             **************************************************************
                             undefined __stdcall entry(void)
                               assume FS_OFFSET = 0xffdff000
             undefined         AL:1           <RETURN>
                             entry                                    XREF[2]:     Entry Point(*), 00400138(*)
        00427dcd e8 b5 d0        CALL       ___security_init_cookie                         void ___security_init_cookie(void)
                 00 00
        00427dd2 e9 7f fe        JMP        FUN_00427c56                                    int FUN_00427c56(void)
                 ff ff
                             -- Flow Override: CALL_RETURN (CALL_TERMINATOR)
        00427dd7 cc              ??         CCh
        00427dd8 cc              ??         CCh
        00427dd9 cc              ??         CCh
        00427dda cc              ??         CCh
        00427ddb cc              ??         CCh
        00427ddc cc              ??         CCh
        00427ddd cc              ??         CCh
        00427dde cc              ??         CCh
        00427ddf cc              ??         CCh

                             **************************************************************
                             * Library Function - Multiple Matches With Different Base... *
                             *  _memcpy                                                   *
                             *  _memmove                                                  *
                             *                                                            *
```

```

void entry(void)

{
  ___security_init_cookie();
  FUN_00427c56();
  return;
}

```

Figure 2.6.5 - Ghidra’s Main Interface and Entry point

Clicking this entry point will direct the Listing to the appropriate line where we can see the immediate call of the __security_init_cookie function, a C function used for protection against buffer overrun (https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/security-init-cookie last day visited - 14/2/2023). And since Assembly executes lines consecutively (unless stated otherwise with a jump instruction), the next operation is executed. On this particular occasion, there is a jump instruction leading to a function named FUN_00427c56 and by double clicking on its name, the Listing view changes to display that particular function and we can see the code decompiled to C language at the right panel.



```
00427c39 83 c6 02        ADD      ESI,0x2
00427c3c 0f b7 16        MOVZX    EDX,word ptr [ESI]
00427c3f 66 85 d2        TEST     DX,DX
00427c42 75 f0           JNZ      LAB_00427c34

                    LAB_00427c44                          XREF[1]:     00427c30(j)
00427c44 83 c0 02        ADD      EAX,0x2
00427c47 0f b7 08        MOVZX    ECX,word ptr [EAX]
00427c4a 66 85 c9        TEST     CX,CX
00427c4d 75 dc           JNZ      LAB_00427c2b

                    LAB_00427c4f                          XREF[1]:     00427c23(j)
00427c4f 33 c0           XOR      EAX,EAX

                    LAB_00427c51                          XREF[1]:     00427c37(j)
00427c51 5f              POP      EDI
00427c52 5e              POP      ESI
00427c53 5b              POP      EBX
00427c54 5d              POP      EBP
00427c55 c3              RET

                    ****************************************************************
                    *                           FUNCTION                           *
                    ****************************************************************
                    int __stdcall FUN_00427c56(void)
                      assume FS_OFFSET = 0xffdff000
     int              EAX:4          <RETURN>
     undefined4       Stack[-0x8]:4  local_8                     XREF[2]:     00427ce1(RW),
                                                                              00427d97(W)
     undefined4       Stack[-0x20]:4 local_20                    XREF[1]:     00427cb7(W)
     undefined4       Stack[-0x28]:4 local_28                    XREF[1]:     00427d55(W)
                    FUN_00427c56                          XREF[1]:     entry:00427dd2(c)
00427c56 6a 14           PUSH     0x14
00427c58 68 b8 a0        PUSH     DAT_004ba0b8                            = FEh
         4b 00
00427c5d e8 9e 0c        CALL     __SEH_prolog4                           undefined __SEH_prolog4(undefine...
         00 00
00427c62 e8 a1 21        CALL     FUN_00429e08                            WORD FUN_00429e08(void)
         00 00
00427c67 0f b7 f0        MOVZX    ESI,AX
00427c6a 6a 02           PUSH     0x2
00427c6c e8 c9 d1        CALL     FUN_00434e3a                            undefined FUN_00434e3a(undefined...
         00 00
00427c71 59              POP      ECX
00427c72 b8 4d 5a        MOV      EAX,0x5a4d
         00 00
00427c77 66 39 05        CMP      word ptr [IMAGE_DOS_HEADER_00400000],AX
         00 00 40 00
```

```

/* WARNING: Function: __SEH_prolog4 replaced with injecti
/* WARNING: Function: __SEH_epilog4 replaced with injecti
/* WARNING: Removing unreachable block (ram,0x00427c80) *

int FUN_00427c56(void)

{
  bool bVar1;
  undefined3 extraout_var;
  int iVar2;
  VARIANTARG **ppVVar3;

  FUN_00429e08();
  FUN_00434e3a(2);
  bVar1 = FUN_00428b7c();
  if (CONCAT31(extraout_var,bVar1) == 0) {
    _fast_error_exit(0x1c);
  }
  iVar2 = FUN_00429ae6();
  if (iVar2 == 0) {
    _fast_error_exit(0x10);
  }
  FUN_00429bcb();
  iVar2 = FUN_0042d5d2();
  if (iVar2 < 0) {
    _fast_error_exit(0x1b);
  }
  DAT_004c4228 = GetCommandLineW();
  DAT_004c31f0 = FUN_00434f23();
  iVar2 = FUN_00434b1b();
  if (iVar2 < 0) {
    __amsg_exit(8);
  }
  iVar2 = FUN_00434d58();
  if (iVar2 < 0) {
    __amsg_exit(9);
  }
  iVar2 = FUN_004230ef(1);
  if (iVar2 != 0) {
    __amsg_exit(iVar2);
  }
  ppVVar3 = (VARIANTARG **)__wwincmdln();
  iVar2 = FUN_004047d0(0x400000,0,ppVVar3);
  _exit(iVar2);
  __cexit();
  return iVar2;
}
```

Figure 2.6.6 - Viewing the code

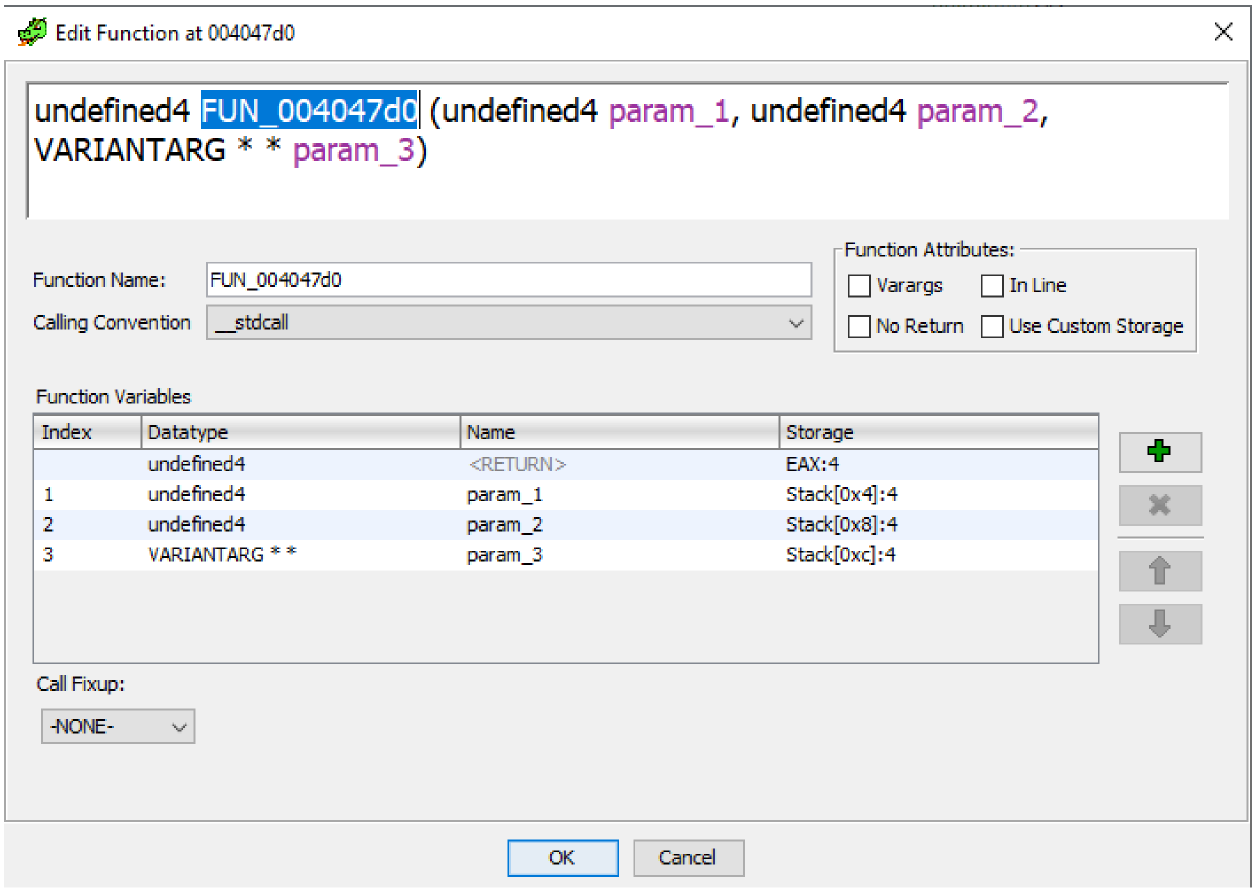
Here, we search for a function call with three arguments. In most cases there will be only one and it will be located just above the exit functions, however we need to check every function with three arguments. According to the documentation (https://learn.microsoft.com/en-us/cpp/cpp/main-function-command-line-args last day visited - 14/2/2023) this is the signature of a Windows application’s main function:

```
int main(int argc, char* argv[], char* envp[]);
```

Double clicking on this function's name will get us to a different section of the code where we can view code fairly familiar with what a human would write.
Now, we can start the slow process of understanding the code. Obviously, each piece of malware is different and there can be no universal procedure, however there are a few techniques we can use to make our life easier.

Editing the Names and the Signatures

The first thing we can do is identify the functions and the variables and edit their name to something we can understand better. For example, we just found the main function, still named FUN_004047d0. By right clicking, we can select "Edit Function Signatures" and change it to the main function's convention we found in the documentation.


Edit Function at 004047d0

```
undefined4 FUN_004047d0 (undefined4 param_1, undefined4 param_2,
VARIANTARG * * param_3)
```

Function Name: FUN_004047d0

Calling Convention: __stdcall

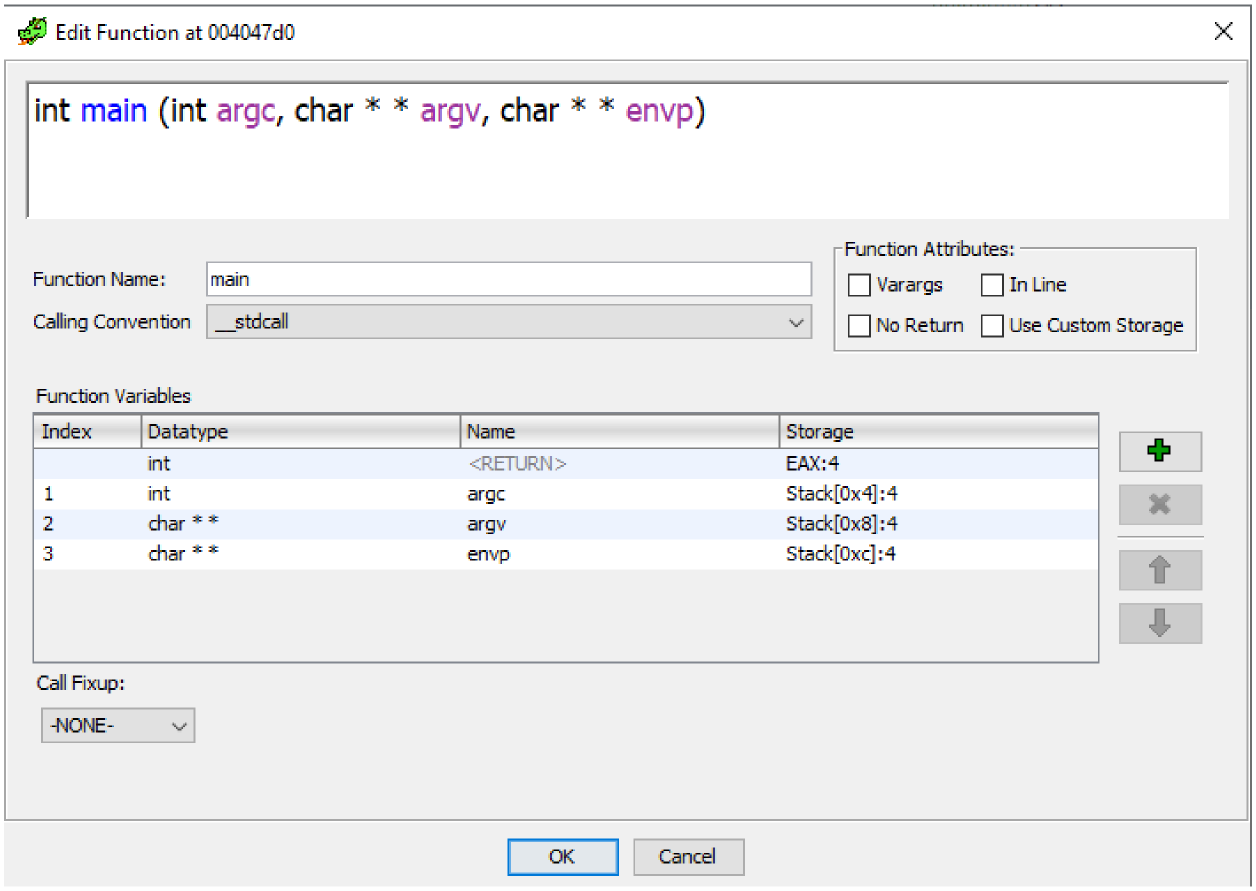
Function Attributes: Varargs ☐ In Line ☐ No Return ☐ Use Custom Storage ☐

Function Variables

Index	Datatype	Name	Storage
	undefined4	<RETURN>	EAX:4
1	undefined4	param_1	Stack[0x4]:4
2	undefined4	param_2	Stack[0x8]:4
3	VARIANTARG * *	param_3	Stack[0xc]:4

Call Fixup: -NONE-

Figure 2.6.7.1 - Before editing the names


Edit Function at 004047d0

```
int main (int argc, char * * argv, char * * envp)
```

Function Name: main

Calling Convention: __stdcall

Function Attributes: ☐ Varargs ☐ In Line ☐ No Return ☐ Use Custom Storage

Function Variables

Index	Datatype	Name	Storage
	int	<RETURN>	EAX:4
1	int	argc	Stack[0x4]:4
2	char * *	argv	Stack[0x8]:4
3	char * *	envp	Stack[0xc]:4

Call Fixup: -NONE-

Figure 2.6.7.2 - After editing the names

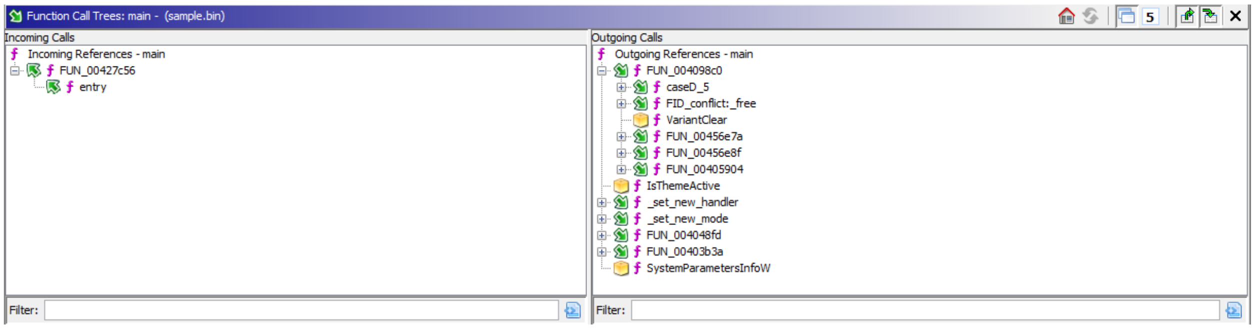
If we look at the Listing now, we will notice that the function is now named “main”. We now need to check each function and repeat the procedure.

Using Ghidra’s Tools

There are various graphs we can use to make this process not easier, but a bit less time consuming.

Function Call Trees

Under the Window menu, there is the option to display the Function Call Trees. This new panel will open at the bottom of the window and display all the calls, both incoming and outgoing, that will be made by the function we are currently in. This tool is useful for lots of reasons, including but not limited to showing us the functions we have not yet identified and renamed, the number of functions that will be called and more.



Function Call Trees: main - (sample.bin)

Incoming Calls

```
Incoming References - main
⊟ FUN_00427c56
   └ entry
```

Filter:

Outgoing Calls

```
Outgoing References - main
⊟ FUN_004098c0
   ⊞ caseD_5
   ⊞ FID_conflict:_free
     VariantClear
   ⊞ FUN_00456e7a
   ⊞ FUN_00456e8f
   ⊞ FUN_00405904
  IsThemeActive
⊞ _set_new_handler
⊞ _set_new_mode
⊞ FUN_004048fd
⊞ FUN_00403b3a
  SystemParametersInfoW
```

Filter:

Figure 2.6.8 - The Function Call Trees Panel

Function Call Graph

The Function Call Graph is complementary to the Function Call Trees and provides a useful and convenient way to visualize the malware's flow. Each node is clickable and by pressing the plus sign we can view the new function's calls.

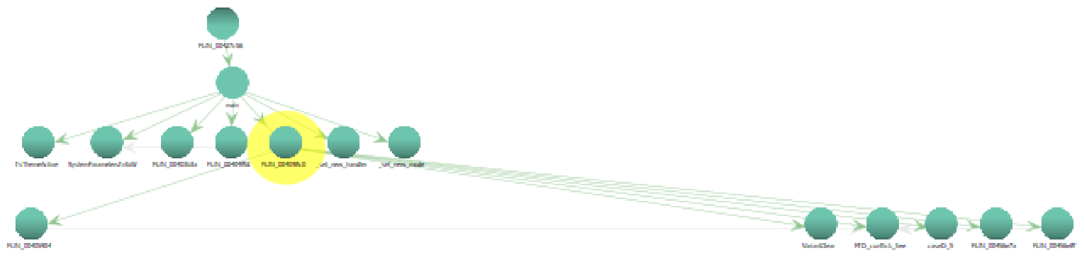
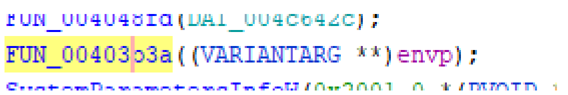


Figure 2.6.9 - The Function Call Graph Window

Searching for Suspicious Functionality

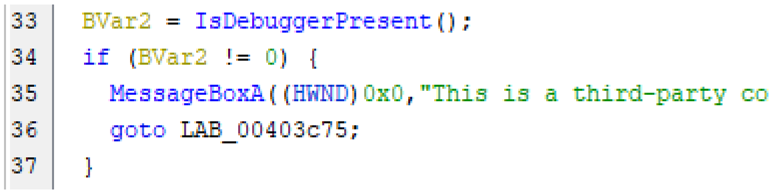
The main reason I specifically picked a trojan horse for this section is because we must understand that there will be legitimate code there as part of the host software. That code should remain intact and we should look for anything out of the ordinary. One of the function calls that stand out in main is a function named "FUN_00403b3a". The reason it stands out is because it directly takes as an argument one of the three arguments that main has. Does that mean that this call is malicious in nature? Not really, no. But we are in the process of investigation and thus we should really check it out.



```
FUN_004048fd(DAT_004c642c);
FUN_00403b3a((VARIANTARG **)envp);
```

Figure 2.6.9 - Investigating function calls

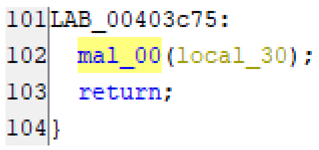
Double clicking on this function will navigate us to the appropriate section of the code. As a reminder, this code could be completely legitimate. In this case though, there is an extremely suspicious condition early on, starting in line 33 up to line 37.



```
33  BVar2 = IsDebuggerPresent();
34  if (BVar2 != 0) {
35    MessageBoxA((HWND)0x0,"This is a third-party co
36    goto LAB_00403c75;
37  }
```

Figure 2.6.10 - Investigating suspicious conditions

This is something we delved into before, in the Malware Analysis chapter, where we displayed the lengths that malware authors are willing to go in order for the malware to go unnoticed. These lines of code exist to check if a debugger is present. If there is, the condition is not met and the rest of the program is executed normally. If there is not, however, the condition is met and the computer is instructed to go to the label “LAB_00403c75” at line 101 (of the same script). There, the function “FUN_00405904” is called. It is safe to assume that this is the malicious section of the code and we can edit its name to reflect so.



```
101 LAB_00403c75:
102   mal_00(local_30);
103   return;
104 }
```

Figure 2.6.11 - Finding the malicious code

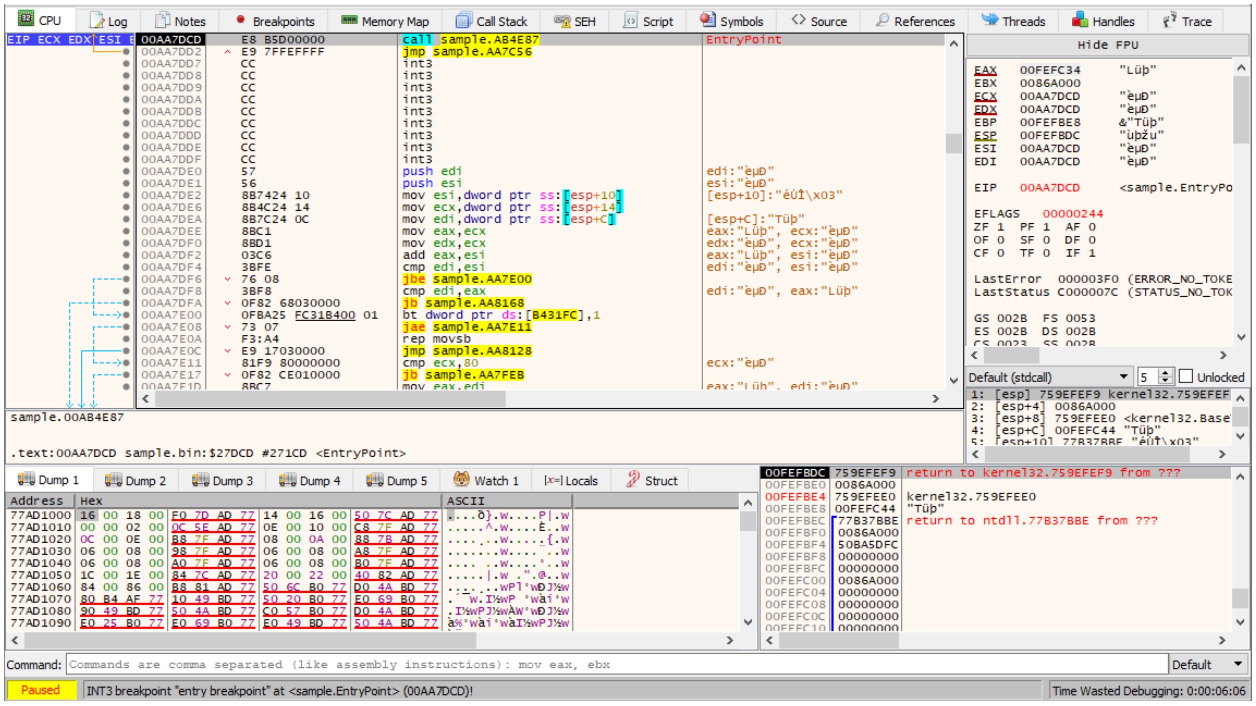
This might be the entire payload, but it might as well be a function that will manipulate the memory or load another sequence of functions or even direct the flow of execution to an entirely different section.

Manipulating Return Values with XDBG

Earlier we saw a condition where the malware is utilizing the IsDebuggerPresent() function to decide whether or not to execute the payload, according to the debugger status on our machine. But what if we want to execute it while there is a debugger? This is where the XDBG comes in. The x64dbg (for 64 bit applications) and the x32dbg (for 32 bit applications) are both included in Flare VM and can be run in conjunction with Ghidra to manipulate in real time the value a function would return as well as the value of any variables.

First Steps

Since Lokibot is a 32 bit malware, we will start x32dbg and open the bin file. Since we only want custom breakpoints, we will head to Settings, under the Options menu, and uncheck System Breakpoint and TLS Callbacks. The System Breakpoint is hardcoded into the ntdll and will trigger once the dlls are loaded. TLS (Thread Local Storage) Callbacks are functions that are called as soon as a new thread is created. We want more strict control over our break points, that is why we disabled both of these. We allow the Entry Breakpoint as a means to conveniently skip to the start of the execution order.



```
EIP ECX EDX ESI
00AA7DCD   E8 B5D00000     call sample.AB4E87                EntryPoint
00AA7DD2   E9 7FFEFFFF     jmp sample.AA7C56
00AA7DD7   CC              int3
00AA7DD8   CC              int3
00AA7DD9   CC              int3
00AA7DDA   CC              int3
00AA7DDB   CC              int3
00AA7DDC   CC              int3
00AA7DDD   CC              int3
00AA7DDE   CC              int3
00AA7DDF   CC              int3
00AA7DE0   57              push edi                          edi:"èµÐ"
00AA7DE1   56              push esi                          esi:"èµÐ"
00AA7DE2   8B7424 10       mov esi,dword ptr ss:[esp+10]     [esp+10]:"éÜÎ\x03"
00AA7DE6   8B4C24 14       mov ecx,dword ptr ss:[esp+14]
00AA7DEA   8B7C24 0C       mov edi,dword ptr ss:[esp+C]      [esp+C]:"Tüþ"
00AA7DEE   8BC1            mov eax,ecx                       eax:"Lüþ", ecx:"èµÐ"
00AA7DF0   8BD1            mov edx,ecx                       edx:"èµÐ", ecx:"èµÐ"
00AA7DF2   03C6            add eax,esi                       eax:"Lüþ", esi:"èµÐ"
00AA7DF4   3BFE            cmp edi,esi                       edi:"èµÐ", esi:"èµÐ"
00AA7DF6   76 08           jbe sample.AA7E00
00AA7DF8   3BF8            cmp edi,eax                       edi:"èµÐ", eax:"Lüþ"
00AA7DFA   0F82 68030000   jb sample.AA8168
00AA7E00   0FBA25 FC31B400 01  bt dword ptr ds:[B431FC],1
00AA7E08   73 07           jae sample.AA7E11
00AA7E0A   F3:A4           rep movsb
00AA7E0C   E9 17030000     jmp sample.AA8128
00AA7E11   81F9 80000000   cmp ecx,80                        ecx:"èµÐ"
00AA7E17   0F82 CE010000   jb sample.AA7FEB
00AA7E1D   8BC7            mov eax,edi                       eax:"Lüþ", edi:"èµÐ"

sample.00AB4E87

.text:00AA7DCD sample.bin:$27DCD #271CD <EntryPoint>
```

```
EAX   00FEFC34     "Lüþ"
EBX   0086A000
ECX   00AA7DCD     "èµÐ"
EDX   00AA7DCD     "èµÐ"
EBP   00FEFBE8     &"Tüþ"
ESP   00FEFBDC     "ùþžu"
ESI   00AA7DCD     "èµÐ"
EDI   00AA7DCD     "èµÐ"

EIP   00AA7DCD     <sample.EntryPo

EFLAGS   00000244
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  000003F0 (ERROR_NO_TOKE
LastStatus C000007C (STATUS_NO_TOK

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B

1: [esp] 759EFEF9 kernel32.759EFEF
2: [esp+4] 0086A000
3: [esp+8] 759EFEE0 <kernel32.Base
4: [esp+C] 00FEFC44 "Tüþ"
5: [esp+10] 77B37BBE "éÜÎ\x03"
```

```
Address   Hex                                               ASCII
77AD1000  16 00 18 00 F0 7D AD 77 14 00 16 00 50 7C AD 77  ....ð}.w....P|.w
77AD1010  00 00 02 00 0C 5E AD 77 0E 00 10 00 C8 7F AD 77  .....^.w....È..w
77AD1020  0C 00 0E 00 B8 7F AD 77 08 00 0A 00 88 7B AD 77  ........w.....{.w
77AD1030  06 00 08 00 98 7F AD 77 06 00 08 00 A8 7F AD 77  .......w......w
77AD1040  06 00 08 00 A0 7F AD 77 06 00 08 00 B0 7F AD 77  .... ..w....°..w
77AD1050  1C 00 1E 00 84 7C AD 77 20 00 22 00 40 82 AD 77  .....|.w .".@..w
77AD1060  84 00 86 00 B8 81 AD 77 50 6C B0 77 D0 4A BD 77  ........wPl°wÐJ½w
77AD1070  80 B4 AF 77 10 49 BD 77 50 20 B0 77 E0 69 B0 77  . ¯w.I½wP °wài°w
77AD1080  90 49 BD 77 50 4A BD 77 C0 57 B0 77 D0 4A BD 77  .I½wPJ½wÀW°wÐJ½w
77AD1090  E0 25 B0 77 E0 69 B0 77 E0 49 BD 77 50 4A BD 77  à%°wài°wàI½wPJ½w
```

```
00FEFBDC  759EFEF9  return to kernel32.759EFEF9 from ???
00FEFBE0  0086A000
00FEFBE4  759EFEE0  kernel32.759EFEE0
00FEFBE8  00FEFC44  "Tüþ"
00FEFBEC  77B37BBE  return to ntdll.77B37BBE from ???
00FEFBF0  0086A000
00FEFBF4  50BA5DFC
00FEFBF8  00000000
00FEFBFC  00000000
00FEFC00  0086A000
00FEFC04  00000000
00FEFC08  00000000
00FEFC0C  00000000
00FEFC10  00000000
```

```
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused  INT3 breakpoint "entry breakpoint" at <sample.EntryPoint> (00AA7DCD)!  Time Wasted Debugging: 0:00:06:06
```

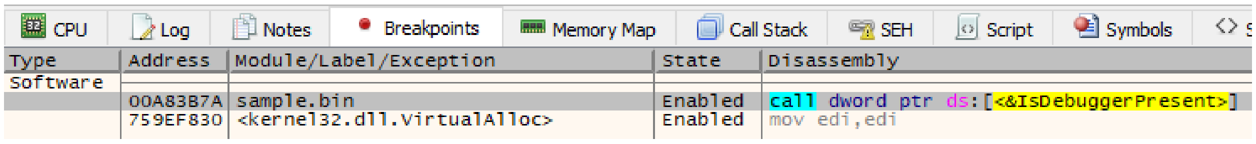
Figure 2.6.12 - The x32dbg interface

Setting the Appropriate Breakpoints

In Ghidra we found that there is an if statement and the condition will be met if a debugger is active. And here, a debugger is obviously active. The first thing we need to do is find the memory address where that if statement takes place and insert a breakpoint there. In order to do so, we must first understand that Ghidra and XDBG will not display the same address for the same statement.

The reason for this is the Address Space Layout Randomization (ASLR), a security feature that randomizes the base memory the application is stored in and thus greatly reduces the capability of malware to perform a buffer overflow. However, since only the base memory is randomized, we can find the call rather easily.

We first notice that in Ghidra the entry point is at the address: “00427dcd” and at xdbg at: “00AA7DCD”. We can immediately understand the correlation as the base address is different but the secondary address is the same. Returning to Ghidra, we can see that the IsDebuggerPresent() call takes place at the address: “00403b7a”. The entry function has a difference of -2 and thus the function at hand should also have a difference of -2. So, we should search at the address: “00A83B7A” on xdbg. Once we find the address, simply pressing F2 will insert a breakpoint (F2 is the default shortcut, we can always right click and at the breakpoints menu press “Toggle”).
The second breakpoint we need to enter is during the VirtualAlloc() call (https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc last day visited - 14/2/2023). VirtualAlloc() is a function used to reserve memory space while the application is already running. While VirtualAlloc() is an important function for legitimate Windows applications, it is very often used by malware. By typing “bp VirtualAlloc” in the command line at the bottom of xdbg, a breakpoint is immediately created wherever this call is made.
It is worth noting that had we used “bp IsDebuggerPresent” at the command line, we would have the exact same result, the reason I used the memory address is because I wanted to demonstrate the manual way to do it.

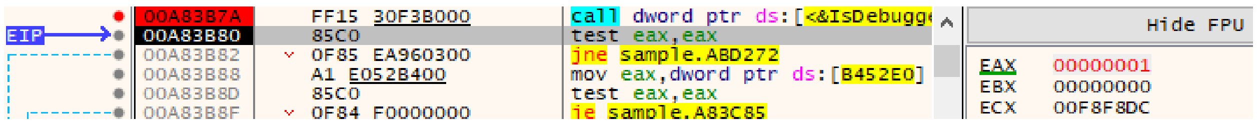


```
Type      Address   Module/Label/Exception                 State     Disassembly
Software
          00A83B7A  sample.bin                             Enabled   call dword ptr ds:[<&IsDebuggerPresent>]
          759EF830  <kernel32.dll.VirtualAlloc>            Enabled   mov edi,edi
```

Figure 2.6.13 - The breakpoints tab

Live Debugging

Now that we know where we need to stop the malware, we can run it from inside the xdbg. Pressing run will execute the file and stop right before calling the IsDebuggerPresent() function. Then, we will go one step over and check the registers.

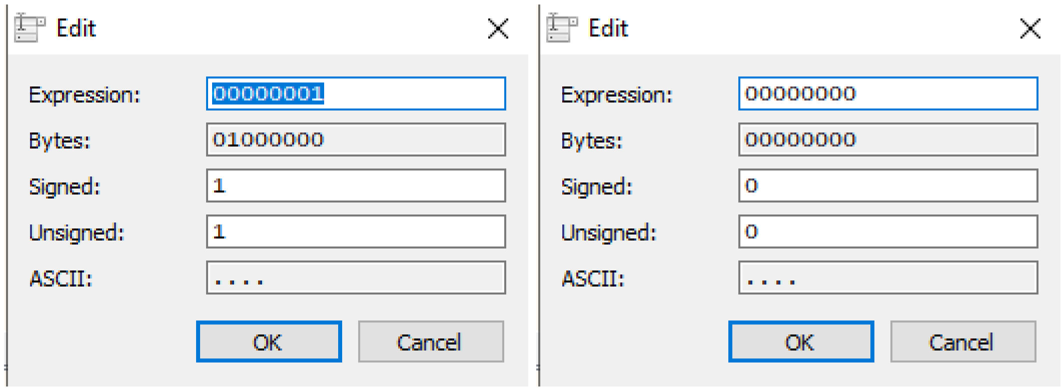


```
00A83B7A   FF15 30F3B000   call dword ptr ds:[<&IsDebugg
00A83B80   85C0            test eax,eax
00A83B82   0F85 EA960300   jne sample.ABD272
00A83B88   A1 E052B400     mov eax,dword ptr ds:[B452E0]
00A83B8D   85C0            test eax,eax
00A83B8F   0F84 F0000000   je sample.A83C85
```

```
EAX   00000001
EBX   00000000
ECX   00F8F8DC
```

Figure 2.6.14.1 - Live modification of registers

As we can see, the Accumulator (EAX register) has a value of 1. This of course, is because a debugger is present. We are now between the call of the function and the condition check. The JNE instruction (stands for Jump Not Equal) will navigate the flow of execution to the malware only if the Accumulator has a value of 0. We can force that by right clicking and selecting “Modify Value”

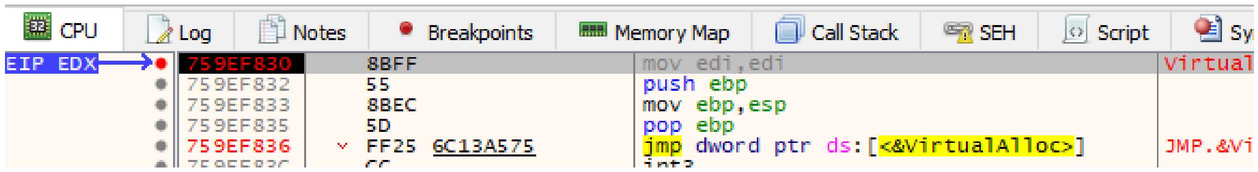


```
Edit
Expression:  00000001
Bytes:       01000000
Signed:      1
Unsigned:    1
ASCII:       ....
```

```
Edit
Expression:  00000000
Bytes:       00000000
Signed:      0
Unsigned:    0
ASCII:       ....
```

Figure 2.6.14.2 - Live modification of registers

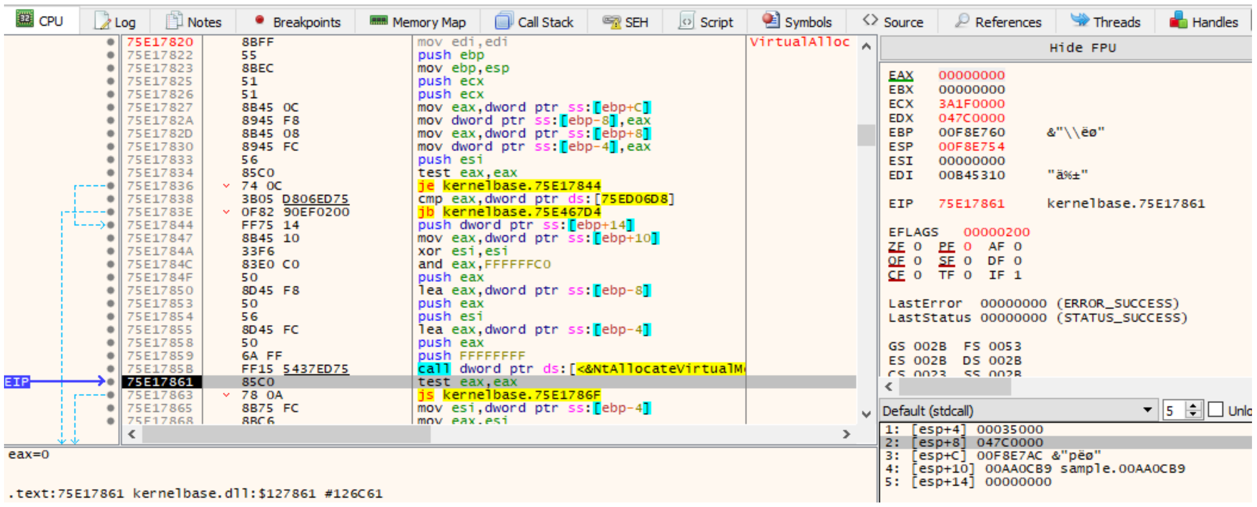
Now, moving forward, the malware will be executed. This is where the second breakpoint comes into effect.
By clicking “Run” once again, the malware will reach the point where VirtualAlloc() is called and xdbg will halt operations. We will have multiple step overs here, as registers are getting the appropriate values, but it is important to move one step at a time.



```
EIP EDX  759EF830   8BFF           mov edi,edi                      Virtual
         759EF832   55             push ebp
         759EF833   8BEC           mov ebp,esp
         759EF835   5D             pop ebp
         759EF836   FF25 6C13A575  jmp dword ptr ds:[<&VirtualAlloc>]  JMP.&Vi
```

Figure 2.6.15 - Running the VirtualAlloc()

Now, we need to follow the operations until we reach another API call named NtAllocateVirtualMemory(). There are two major differences between these two functions. The first has to do with the user. While in VirtualAlloc() the user is the application, in NtAllocateVirtualMemory() the user is the kernel. The second has to do with the way these two functions handle the address of the allocated memory. While VirtualAlloc() stores it in the Accumulator, NtAllocateVirtualMemory() passes it as the second argument, the one we can find at the esp+8 register. (according to the documentation:
https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntallocatevirtualmemory last day visited - 14/2/2023).



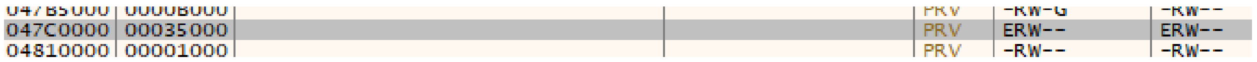
```
75E17820   8BFF            mov edi,edi                                  VirtualAlloc
75E17822   55              push ebp
75E17823   8BEC            mov ebp,esp
75E17825   51              push ecx
75E17826   51              push ecx
75E17827   8B45 0C         mov eax,dword ptr ss:[ebp+C]
75E1782A   8945 F8         mov dword ptr ss:[ebp-8],eax
75E1782D   8B45 08         mov eax,dword ptr ss:[ebp+8]
75E17830   8945 FC         mov dword ptr ss:[ebp-4],eax
75E17833   56              push esi
75E17834   85C0            test eax,eax
75E17836   74 0C           je kernelbase.75E17844
75E17838   3B05 D806ED75   cmp eax,dword ptr ds:[75ED06D8]
75E1783E   0F82 90EF0200   jb kernelbase.75E467D4
75E17844   FF75 14         push dword ptr ss:[ebp+14]
75E17847   8B45 10         mov eax,dword ptr ss:[ebp+10]
75E1784A   33F6            xor esi,esi
75E1784C   83E0 C0         and eax,FFFFFFC0
75E1784F   50              push eax
75E17850   8D45 F8         lea eax,dword ptr ss:[ebp-8]
75E17853   50              push eax
75E17854   56              push esi
75E17855   8D45 FC         lea eax,dword ptr ss:[ebp-4]
75E17858   50              push eax
75E17859   6A FF           push FFFFFFFF
75E1785B   FF15 5437ED75   call dword ptr ds:[<&NtAllocateVirtualM
EIP  75E17861   85C0       test eax,eax
75E17863   78 0A           js kernelbase.75E1786F
75E17865   8B75 FC         mov esi,dword ptr ss:[ebp-4]
75E17868   8BC6            mov eax,esi

eax=0

.text:75E17861 kernelbase.dll:$127861 #126C61
```

```
EAX    00000000
EBX    00000000
ECX    3A1F0000
EDX    047C0000
EBP    00F8E760     &"\\ëø"
ESP    00F8E754
ESI    00000000
EDI    00B45310     "ä%±"

EIP    75E17861     kernelbase.75E17861

EFLAGS    00000200
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B
```

```
1: [esp+4] 00035000
2: [esp+8] 047C0000
3: [esp+C] 00F8E7AC &"pëø"
4: [esp+10] 00AA0CB9 sample.00AA0CB9
5: [esp+14] 00000000
```

Figure 2.6.16 - Getting the Malware's Memory Address

As seen in the Figures 2.6.15 and 2.6.16, we can now get the memory address that the malware was unpacked and by going to the Memory Map tab, we can see that what is there is executable.

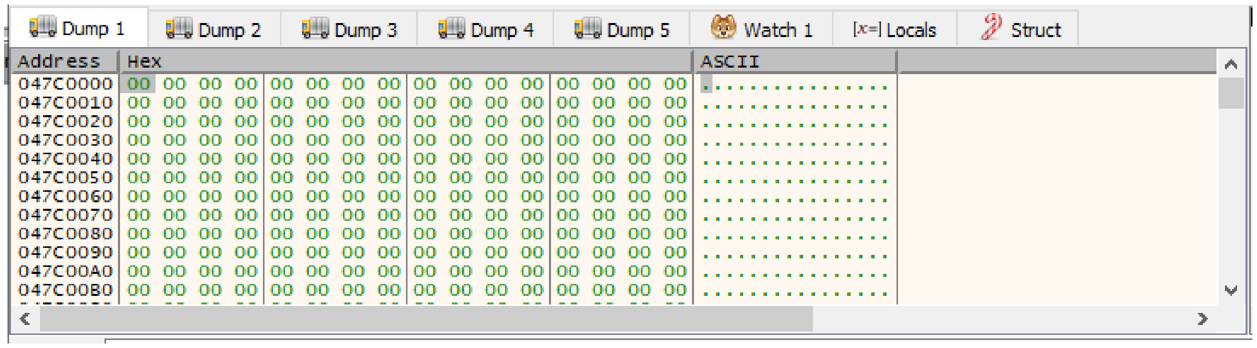


```
047B5000 | 0000B000 |     PRV | -RW-G | -RW--
047C0000 | 00035000 |     PRV | ERW-- | ERW--
04810000 | 00001000 |     PRV | -RW-- | -RW--
```

Figure 2.6.17 - Viewing the Memory Map

Dumping the Memory

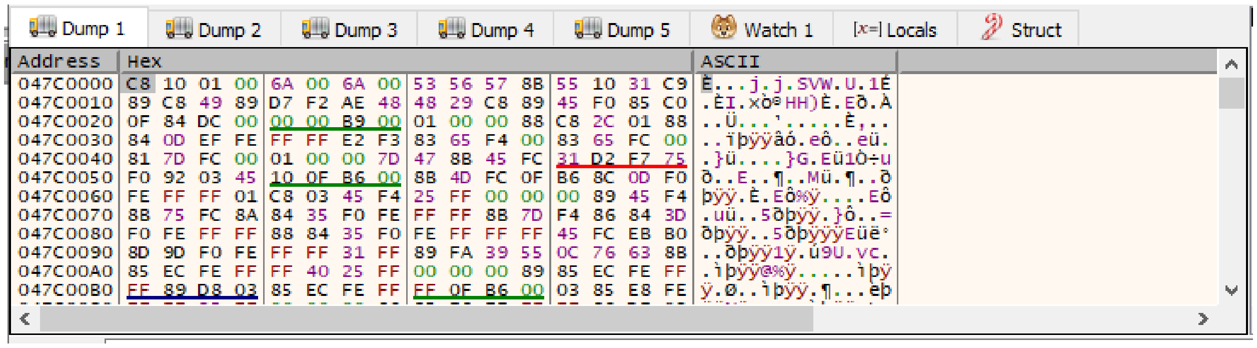
Finally, by right clicking on the address we can select "Follow in Dump" and view whatever is stored in the memory. Since the space was just allocated, it will be filled with zeros, but that will change as soon as the malware runs.



```
Address  | Hex                                             | ASCII
047C0000 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0010 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0020 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0030 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0040 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0050 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0060 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0070 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0080 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C0090 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C00A0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
047C00B0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
```

Figure 2.6.18 - Viewing the freshly Allocated Memory

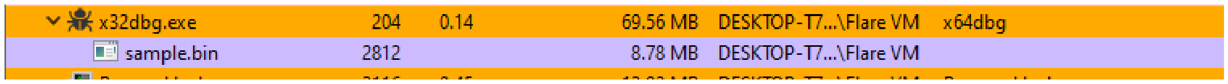
And while we do want it to be unpacked, we want to be notified first. To do that, we will add a hardware breakpoint by right clicking on the first hex location and from the breakpoint menu select “Hardware, Execute”. When we press run again, the entire block will be populated with the malware, just before execution.



```
Address  Hex                                              ASCII
047C0000 C8 10 01 00 6A 00 6A 00 53 56 57 8B 55 10 31 C9  È...j.j.SVW.U.1É
047C0010 89 C8 49 89 D7 F2 AE 48 48 29 C8 89 45 F0 85 C0  .ÈI.xò®HH)È.Eð.À
047C0020 0F 84 DC 00 00 00 B9 00 01 00 00 88 C8 2C 01 88  ..Ü...¹.....È,..
047C0030 84 0D EF FE FF FF E2 F3 83 65 F4 00 83 65 FC 00  ..ïþÿÿâó.eô..eü.
047C0040 81 7D FC 00 01 00 00 7D 47 8B 45 FC 31 D2 F7 75  .}ü....}G.Eü1Ò÷u
047C0050 F0 92 03 45 10 0F B6 00 8B 4D FC 0F B6 8C 0D F0  ð..E..¶..Mü.¶..ð
047C0060 FE FF FF 01 C8 03 45 F4 25 FF 00 00 00 89 45 F4  þÿÿ.È.Eô%ÿ....Eô
047C0070 8B 75 FC 8A 84 35 F0 FE FF FF 8B 7D F4 86 84 3D  .uü..5ðþÿÿ.}ô..=
047C0080 F0 FE FF FF 88 84 35 F0 FE FF FF FF 45 FC EB B0  ðþÿÿ..5ðþÿÿÿEüë°
047C0090 8D 9D F0 FE FF FF 31 FF 89 FA 39 55 0C 76 63 8B  ..ðþÿÿ1ÿ.ú9U.vc.
047C00A0 85 EC FE FF FF 40 25 FF 00 00 00 89 85 EC FE FF  .ìþÿÿ@%ÿ.....ìþÿ
047C00B0 FF 89 D8 03 85 EC FE FF FF 0F B6 00 03 85 E8 FE  ÿ.Ø..ìþÿÿ.¶...èþ
```

Figure 2.6.19 - Viewing the Populated Memory Block

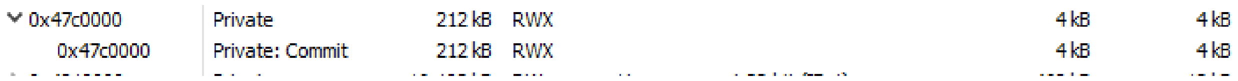
Now, we are going to use a tool familiar from the Malware Analysis chapter, a tool named Process Hacker. Identifying the process is easy, since it will be listed under xdbg.



```
x32dbg.exe      204    0.14    69.56 MB   DESKTOP-T7...\Flare VM    x64dbg
  sample.bin    2812            8.78 MB   DESKTOP-T7...\Flare VM
```

Figure 2.6.20 - Identifying the malware

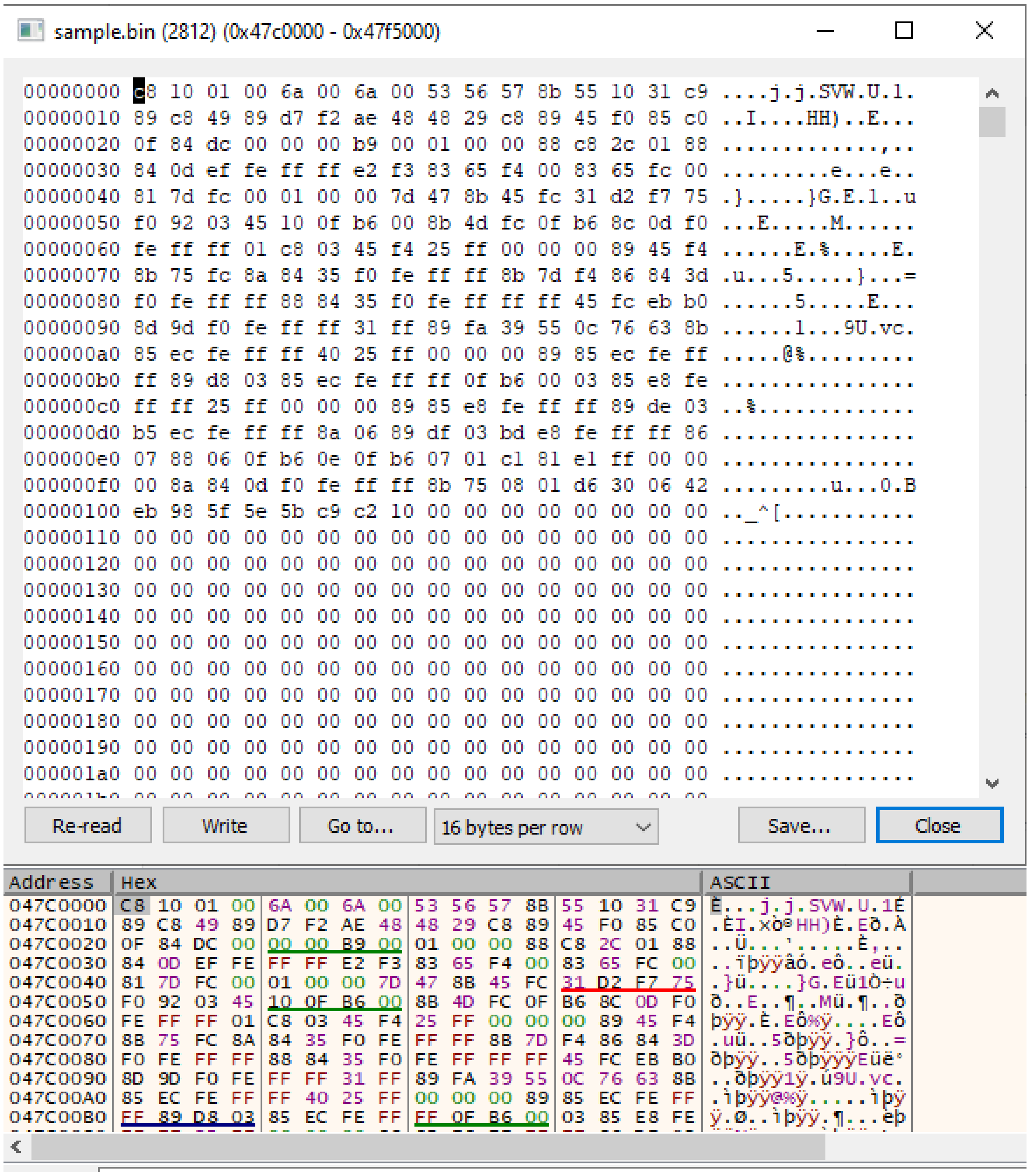
By double clicking on it, a new window will appear and if we go to the memory tab we will be able to find the address that still contains the unpacked but not executed malware.



```
0x47c0000     Private           212 kB   RWX     4 kB    4 kB
  0x47c0000   Private: Commit   212 kB   RWX     4 kB    4 kB
```

Figure 2.6.21 - The Malware’s Memory Address in Process Hacker

As we can see, the permissions include the X tag, indicating that whatever is in that memory block is executable. By double clicking on the expanded item, we can view the contents of this memory block and confirm that it is the same content as it appears in the xdbg panel.



```
00000000 c8 10 01 00 6a 00 6a 00 53 56 57 8b 55 10 31 c9 ....j.j.SVW.U.1.
00000010 89 c8 49 89 d7 f2 ae 48 48 29 c8 89 45 f0 85 c0 ..I....HH)..E...
00000020 0f 84 dc 00 00 00 b9 00 01 00 00 88 c8 2c 01 88 .............,..
00000030 84 0d ef fe ff ff e2 f3 83 65 f4 00 83 65 fc 00 .........e...e..
00000040 81 7d fc 00 01 00 00 7d 47 8b 45 fc 31 d2 f7 75 .}.....}G.E.1..u
00000050 f0 92 03 45 10 0f b6 00 8b 4d fc 0f b6 8c 0d f0 ...E.....M......
00000060 fe ff ff 01 c8 03 45 f4 25 ff 00 00 00 89 45 f4 ......E.%.....E.
00000070 8b 75 fc 8a 84 35 f0 fe ff ff 8b 7d f4 86 84 3d .u...5.....}...=
00000080 f0 fe ff ff 88 84 35 f0 fe ff ff ff 45 fc eb b0 ......5.....E...
00000090 8d 9d f0 fe ff ff 31 ff 89 fa 39 55 0c 76 63 8b ......1...9U.vc.
000000a0 85 ec fe ff ff 40 25 ff 00 00 00 89 85 ec fe ff .....@%.........
000000b0 ff 89 d8 03 85 ec fe ff ff 0f b6 00 03 85 e8 fe ................
000000c0 ff ff 25 ff 00 00 00 89 85 e8 fe ff ff 89 de 03 ..%.............
000000d0 b5 ec fe ff ff 8a 06 89 df 03 bd e8 fe ff ff 86 ................
000000e0 07 88 06 0f b6 0e 0f b6 07 01 c1 81 e1 ff 00 00 ................
000000f0 00 8a 84 0d f0 fe ff ff 8b 75 08 01 d6 30 06 42 .........u...0.B
00000100 eb 98 5f 5e 5b c9 c2 10 00 00 00 00 00 00 00 00 .._^[...........
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
Address  | Hex                                             | ASCII
047C0000 | C8 10 01 00 6A 00 6A 00 53 56 57 8B 55 10 31 C9 | È...j.j.SVW.U.1É
047C0010 | 89 C8 49 89 D7 F2 AE 48 48 29 C8 89 45 F0 85 C0 | .ÈI.xò®HH)È.Eð.À
047C0020 | 0F 84 DC 00 00 00 B9 00 01 00 00 88 C8 2C 01 88 | ..Ü...'.....È,..
047C0030 | 84 0D EF FE FF FF E2 F3 83 65 F4 00 83 65 FC 00 | ..ïþÿÿâó.eô..eü.
047C0040 | 81 7D FC 00 01 00 00 7D 47 8B 45 FC 31 D2 F7 75 | .}ü....}G.Eü1Ò÷u
047C0050 | F0 92 03 45 10 0F B6 00 8B 4D FC 0F B6 8C 0D F0 | ð..E..¶..Mü.¶..ð
047C0060 | FE FF FF 01 C8 03 45 F4 25 FF 00 00 00 89 45 F4 | þÿÿ.È.Eô%ÿ....Eô
047C0070 | 8B 75 FC 8A 84 35 F0 FE FF FF 8B 7D F4 86 84 3D | .uü..5ðþÿÿ.}ô..=
047C0080 | F0 FE FF FF 88 84 35 F0 FE FF FF FF 45 FC EB B0 | ðþÿÿ..5ðþÿÿÿEüë°
047C0090 | 8D 9D F0 FE FF FF 31 FF 89 FA 39 55 0C 76 63 8B | ..ðþÿÿ1ÿ.ú9U.vc.
047C00A0 | 85 EC FE FF FF 40 25 FF 00 00 00 89 85 EC FE FF | .ìþÿÿ@%ÿ.....ìþÿ
047C00B0 | FF 89 D8 03 85 EC FE FF FF 0F B6 00 03 85 E8 FE | ÿ.Ø..ìþÿÿ.¶...èþ
```

Figure 2.6.22 - Comparing the Contents of the Memory Block in Process Hacker (top) with xdbg (bottom)

Now, clicking “Save” on this panel will let us save the binary in its isolated form. This stripped down version is only the malicious code and it can be further analyzed by all the techniques discussed in the Malware Analysis chapter.

Ethical Concerns

We saw how to analyze malware, deobfuscate code and detonate malware in a safe environment as well as the way to isolate a piece of malware from its carrier program. In this chapter, we need to back off from the technical aspect of malware analysis and reverse engineering and take a more philosophical approach. Ethics are not discussed adequately in cybersecurity and this specific field does provide some serious issues to be concerned about.

1. Privacy: Malware analysis often involves analyzing and potentially sharing sensitive personal or confidential information that could be contained within, or targeted by, the malware. This raises concerns about protecting the privacy of individuals whose information may be exposed during the analysis process. This can include financial information, passwords, and other confidential data. To address this concern, analysts should take care to handle and store sensitive data in a secure and confidential manner.
2. Legal issues:There is always the chance for analysts to perform illegal activities while trying to simply work on their occupation. This can include accessing and analyzing systems or data without proper authorization, causing involuntary harm to a system or even violating laws regarding the use of copyrighted material. It is vital for analysts to be aware of the legal framework, the implications of their work and the boundaries of the law they are subjected to and need to operate on.
3. Professional ethics: Analysts have a professional responsibility to conduct their work in an ethical and responsible manner. This includes being honest and transparent about their methods and findings, and being mindful of the potential impact of their work on others. On the same note and part of the Professional ethics, is the proper use of tools and techniques and this is applied to both Malware Analysis and Reverse Engineering.
    a. Malware analysis tools and techniques can be used for malicious purposes, such as to create or disseminate malware. Analysts must be careful to use these tools and techniques ethically and only for defensive purposes.
    b. Reverse Engineering tools and techniques can be used to understand legitimate proprietary software and understand part of the code for use in an unrelated manner at best or corporate espionage at worst. Analysts should remain neutral and only reverse engineer malware for security purposes.
4. Attribution: It can be difficult to determine the true origin and purpose of a piece of malware, and attribution errors can have serious consequences. Analysts must work along with law enforcement and be very cautious to accurately

attribute malware to the appropriate actor, whether that be a single individual or a hacking group, and not make assumptions based on incomplete or biased evidence.

5. Vulnerability disclosure: Analysts who discover vulnerabilities in software as part of their malware analysis must decide not if but how to disclose these vulnerabilities to the affected parties. This process can be complex and must be done with care to avoid causing harm. This is the reason major organizations have a vulnerability disclosure program and even pay the so-called “bounty hunters” for utilizing it.

The Future of Malware Analysis and Reverse Engineering

As we progress into an even more digitalized era of human history, we can assume with relative certainty that malware will also continue to evolve. New adversaries will replace the departing ones and with the new generation being so technologically adept, they will surely pose a threat, a caliber of which we have not yet faced.

What we can do, as members of the Cybersecurity community that strive for safer and more secure utilizations of technology, is evolve with the threats. This evolution will be led by professionals who excel in new technologies while humble enough to be mentored by the old guard.

These new technologies revolve around Machine Learning, as with everything these days, which is maturing to a level hardly imaginable just a few years ago. Machine Learning algorithms can automate several tasks like identifying patterns in memory management and code execution, detect inconsistencies in network traffic and singling out anomalies in API calls.

Furthermore, Machine Learning algorithms can be used to deobfuscate code, provide the source and even identify the malicious actors. In the case of malware being served hidden in another legitimate program, some algorithms could be trained to strip and isolate the malicious binaries while keeping the legitimate software intact.

Advanced ML algorithms could potentially streamline the entire process and produce results in both known and novelty threats, while keeping the system safe by the use of temporary hypervisor technology.

However, we are not there just yet. The main reason is the huge amount of data such a sophisticated system would need to be trained on and this amount of data is not widely available. And still, despite the challenge, Machine Learning has the potential to revolutionize the entire Cybersecurity sector and especially the fields of Malware Analysis and Reverse Engineering and lead to a technologically safer future.

Sources

[1] Jordan Spencer Cunningham, 2016, ‘Interview with Ray Tomlinson on Creeper/Reaper’, OSNews.com, 2016/04/06
[2] John Walker, 1996, ‘The Animal Episode’, fourmilab.ch, 1996/08/21
[3] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim, 2015, ‘A Novel Approach to Detect Malware Based on API Call Sequence Analysis’, Sage Journals
[4] Benjamin 2014, ‘Passwords’, Internet Source
[5] Ashkan Hosseini, 2017, ‘Ten process injection techniques: A technical survey of common and trending process injection techniques’, Internet Source